Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Tvorba webových aplikací v ASP.NET

Bakalářská práce

Autor práce: Ondřej Svačina

Vedoucí práce: Ing. Jiří Brožek, Ph.D.

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Ondřej Svačina

Informatika

Název práce

Tvorba webových aplikací v ASP.NET

Název anglicky

Web application development using ASP.NET

Cíle práce

Práce je zaměřena na problematiku tvorby webových aplikací s využitím ASP.NET. Hlavním cílem je prostřednictvím tvorby ukázkové aplikace pracující s databází představit možnosti ASP.NET a souvisejících technologií a frameworků. Dílčím cílem je pak popis jednotlivých technologií.

Metodika

Metodika bakalářské práce je založena na analyticko-syntetickém přístupu. Bude provedeno studium a analýza odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou popsány možnosti ASP.NET a souvisejících technologií při tvorbě webových aplikací. Dále bude proveden návrh a implementace ukázkové webové aplikace pracující s daty v databázi. Postup implementace bude popsán a zhodnocen.

Doporučený rozsah práce

35-40 stran

Klíčová slova

ASP.NET, MVC, Razor, C#, HTML, CSS, Javascript, jQuery

Doporučené zdroje informací

ALBAHARI, Joseph a ALBAHARI, Ben. C# in a nutshell. 5th Edition. Sebastopol: O'Reilly, 2012. ISBN 978-1-449-32010-2

ESPOSITO, Dino. Programming Microsoft ASP.NET MVC. Third Edition. Sebastopol: O'Reilly, 2014. ISBN 978-0-7356-8094-4

FOWLER, Martin, RICE, David, FOEMMEL, Matthew, HIEATT, Edward, MEE, Robert, STAFFORD, Randy. Patterns of Enterprise Application Architecture. Boston: Addison Wesley, 2002. ISBN 0-321-12742-0

CHADWICK, Jess. Programming Razor. Sebastopol: O'Reilly, 2011. ISBN 978-1-449-30676-2

MUELLER, John Paul. Microsoft ADO.NET Entity Framework Step by Step. Redmond: Microsoft Press, 2013. ISBN 978-0-735-66416-6

Předběžný termín obhajoby

2015/16 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 13. 03. 2016

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Tvorba webových aplikací v ASP.NET" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 13.3.2016 _____

Poděkování

Touto cestou bych rád poděkoval vedoucímu práce Ing. Jiřímu Brožkovi, Ph.D. za hodnotné konzultace a také Ing. Josefu Škeříkovi, CSc. a Ing. Aleně Škeříkové za užitečné rady a připomínky k práci.

Tvorba webových aplikací v ASP.NET

Souhrn

Bakalářská práce se v teoretické části věnuje popisu vlastností a možností webového frameworku ASP.NET společnosti Microsoft, především jeho části ASP.NET MVC. Také se věnuje některým dalším technologiím a frameworkům, jako je Razor, jQuery, Ajax, Bootstrap a Entity Framework, které se při vývoji moderních webových aplikací běžně používají. V praktické části práce je podrobně popsán postup vývoje jednoduché webové MVC aplikace pracující s databází. Je popsáno a vysvětleno praktické použití frameworku MVC, ostatních frameworků a technologií souvisejících s vývojem webových aplikací a také je předvedeno několik postupů a návrhových vzorů často užívaných při vývoji moderních aplikací.

Klíčová slova: web, databáze, ASP.NET, MVC, Razor, C#, Entity Framework

Web application development using ASP.NET

Summary

In its theoretical part, this bachelor thesis describes features and capabilities of Microsoft ASP.NET web framework, in particular one of its parts, ASP.NET MVC. It also deals with some other technologies and frameworks such as Razor, jQuery, Ajax and Entity Framework, which are commonly used in modern web application development. In practical part, the process of developing simple database web application is described in detail. There is described and explained practical use of MVC framework and other frameworks and technologies related to web development. Also, several development approaches and design patterns are demonstrated.

Keywords: web, databases, ASP.NET, MVC, Razor, C#, Entity Framework

Obsah

1. Úvod

S webovými aplikacemi se setkáváme stále častěji. S rozvojem a neustálým zrychlováním internetu, a také s obrovským rozšířením chytrých mobilních telefonů a jiných zařízení napříč celou populací, se zdá skoro nemožné na některou z nich nenarazit. Chceme-li si zakoupit vstupenky do kina, nejjednodušší a nejrychlejší cestou je pomocí webové aplikace. Chceme-li zjistit, v kolik nám jede vlak nebo autobus, opět se nám nabízí jako nejsnazší způsob webová aplikace dopravce, kde jsou navíc informace stále aktuální. Chceme-li se podívat na zůstatek na našem bankovním účtu nebo zaplatit složenku, použijeme webovou aplikaci. Webové aplikace najdou uplatnění snad ve všech možných oblastech. Mají oproti klasickým[1] aplikacím mnoho výhod. Především je uživatel nemusí instalovat do svého počítače, tabletu nebo telefonu. Stačí mu pouze počítač s připojením k internetu a internetový prohlížeč. Obrovskou výhodou je také možnost použití jedné webové aplikace na téměř jakémkoliv zařízení s přístupem k internetu. Je jedno, zda má uživatel počítač, tablet nebo chytrý telefon, nebo který operační systém používá. Vývojář nemusí pro každý přístroj a operační systém vyvíjet samostatnou verzi, tak jako je tomu u klasických aplikací. Myslím, že webové aplikace mají před sebou světlou budoucnost a že budou do našich životů zasahovat stále častěji a výrazněji.

V současné době existuje několik platforem pro vývoj webových aplikací. Já jsem si pro svoji práci vybral ASP.NET od společnosti Microsoft, konkrétně jednu jeho část, a to MVC Framework. Důvodem je rozšíření a upevnění mých znalostí ASP.NET a možnost jejich následného uplatnění v zaměstnání, kde již využívám jinou část ASP.NET, a to framework Web Forms.

[1] Klasickými aplikacemi mám na mysli takové aplikace, které si uživatel stáhne z internetu, nebo nainstaluje z některého přenosného média na harddisk ve svém počítači nebo jiném zařízení.

2. Cíl práce a metodika

2.1 Cíl práce

Práce je zaměřená na problematiku tvorby webových aplikací s využitím webového frameworku ASP.NET společnosti Microsoft, především jedné jeho části, a to ASP.NET MVC frameworku. Cílem práce je pomocí ukázkové webové aplikace pracující s databází demonstrovat praktické využití frameworků ASP.NET a ASP.NET MVC a dalších moderních technologií a frameworků používaných při vývoji moderních webových aplikací. Dílčím cílem práce je popis vlastností a komponent ASP.NET, především ASP.NET MVC a návrhového vzoru *model-view-controller*, ze kterého vychází, a také popis některých dalších frameworků a technologií použitých v ukázkové aplikaci.

2.2 Metodika práce

Metodika práce je založena na analyticko-syntetickém přístupu. Bude provedeno studium a analýza tištěných i elektronických odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou v teoretické části práce popsány jednotlivé technologie a frameworky využívané při vývoji moderních webových ASP.NET MVC aplikací. V praktické části aplikace bude proveden návrh a popis implementace aplikace. Aplikace bude vyvíjena ve vývojovém prostředí Visual Studio Community. Nejprve bude popsán účel aplikace a definovány požadavky na chování. Dále bude proveden základní návrh architektury a pomocí wireframů bude proveden návrh grafického uživatelského rozhraní. Samotná implementace začne přípravou řešení, instalací potřebných frameworků a knihoven a vytvořením základní infrastruktury aplikace. Jednotlivé kroky postupu vývoje budou popsány a některé složitější nebo méně jasné programové konstrukce budou vysvětleny. Popis postupu vývoje bude pro lepší pochopení doplňován výpisy s částmi kódu aplikace, popřípadě obrázky. V průběhu vývoje aplikace také budou psány unit testy. Avšak vzhledem k tomu, že unit testů bývá mnoho, jsou často rozsáhlé a v průběhu vývoje se mohou měnit a také nejsou funkční součástí kódu aplikace, nebudou uváděny přímo v textu praktické části, ale budou všechny uvedeny v příloze této práce. Nakonec bude navrženo další směřování a možnosti vývoje vytvořené aplikace.

3. Přehled řešené problematiky

3.1 Microsoft ASP.NET

ASP.NET byl v primárně navržen jako *server-side* programovací platforma. To znamená, že veškerý ASP.NET kód běží na straně webového serveru. V okamžiku, kdy běh ASP.NET kódu skončí, webový server odešle uživateli konečný výsledek – obyčejnou HTML stránku, která může být zobrazena v jakémkoliv webovém prohlížeči (MacDonald, 2012, s. 6). ASP.NET je součástí .NET frameworku a programátor má při programování v ASP.NET přístup ke třídám definovaným v .NET. Může psát aplikace v jakémkoliv jazyce kompatibilním s CLR (common language runtime), včetně jazyků Microsoft Visual Basic a C#. ASP.NET nabízí tři frameworky pro tvorbu webových aplikací: ASP.NET Web forms, ASP.NET MVC a ASP.NET Web Pages (Microsoft, nedatováno).

3.2 ASP.NET Web forms

Jeden z frameworků platformy ASP.NET. Web Forms jsou stránky, které si uživatel může vyžádat skrze internetový prohlížeč. Tyto stránky se mohou skládat z kombinace HTML kódu, klientských skriptů a serverového kódu. Když si uživatel vyžádá stránku, ta je na serveru pomocí frameworku zkompilována a následně vygenerována jako HTML stránka, kterou může zobrazit internetový prohlížeč (Microsoft, nedatováno).

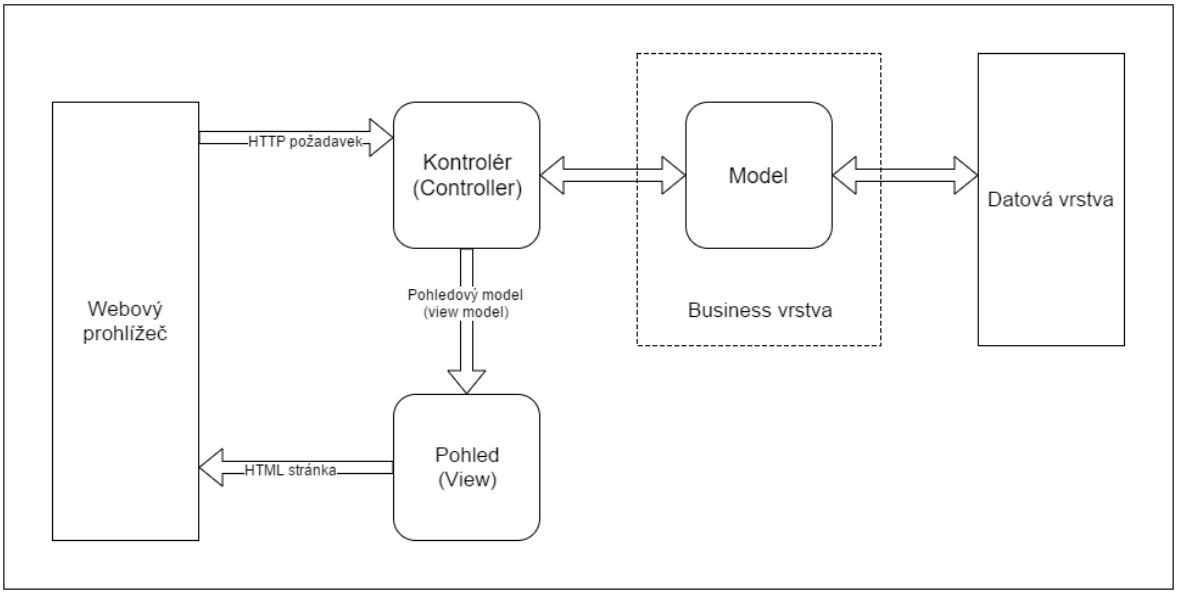
3.3 Návrhový vzor Model-View-Controller (MVC)

Výraz Model-View-Controller existuje již od pozdních 70. let 20. století a poprvé byl použit ve Smalltalk projektu společnosti Xerox PARC, kde byl koncipován jako způsob organizování raných GUI aplikací. Návrhový vzor MVC vynucuje *oddělení odpovědností (separation of concerns)* – doménový model a logika kontroléru jsou oddělené od uživatelského rozhraní. Ve webové aplikaci to znamená, že HTML je oddělené od zbytku aplikace, což umožňuje spravování a testování aplikace jednodušší a snazší (Freeman, 2013).

Fowler et al. (2002) definují tři části MVC vzoru následovně: „Model je objekt, který reprezentuje nějakou informaci o doméně. Je to nevizuální objekt obsahující všechna data a chování, kromě takových, která jsou používána pro uživatelské rozhraní. Pohled (View) reprezentuje zobrazení modelu v uživatelském rozhraní. Pohled je zodpovědný pouze za zobrazení informace a jakákoliv změna v informaci je zpracována třetím členem MVC

vzoru: kontrolérem. Kontrolér přijímá vstup od uživatele, manipuluje s modelem a způsobuje, že se pohled podle provedených změn aktualizuje.“ Na obrázku 3-1 zobrazeno interakcí mezi kontrolérem, modelem a pohledem a také interakcí s vnějším světem a ostatními částmi aplikace.

Obrázek 3-1: Schéma MVC vzoru. *Zdroj: vlastní zpracování*



Je důležité říci, že návrhový vzor Model-View-Controller je architektonický vzor popisující strukturu prezentační vrstvy a ne celé aplikace. Galloway (2014) píše, že MVC představuje řešení pro interakci mezi aplikací a uživatelem, ale neříká nic o tom, jak se řeší ostatní záležitosti aplikace, jako je přístup k datům, interakce se službami atd.

S MVC návrhovým vzorem se lze setkat ve velkém množství programovacích jazyků a v různých platformách. Vzor byl použit v nesčetném množství frameworků a jedním z nich je framework ASP.NET MVC od společnosti Microsoft, který popíši detailněji dále.

3.4 ASP.NET MVC 5 Framework

„ASP.NET MVC je webový vývojový Framework od společnosti Microsoft, který kombinuje efektivnost a pořádek MVC návrhového vzoru, většinu současných myšlenek a technik agilního vývoje a nejlepší součásti platformy ASP.NET. Je to úplná alternativa k tradičnímu ASP.NET Web Forms frameworku přinášející výhody ve všech, kromě těch úplně nejjednodušších, webových projektech“ (Freeman, 2013).

3.4.1 Kontroléry a akční metody

Podle Gallowaye (2014) jsou kontroléry v rámci MVC odpovědné za reagování na vstup uživatele, často formou působení změn v modelu. Mají na starosti „provoz“ v aplikaci, pracují s přicházejícími daty a odchozí data předávají pohledům, kterým jsou určena. Freeman (2013) píše, že „každý požadavek, který dorazí do aplikace, je zpracován kontrolérem. Kontrolér může s požadavkem nakládat tak, jak uzná za vhodné, pokud nebude působit v poli působnosti modelu nebo pohledu. To znamená, že kontroléry nedrží žádná data ani nevytvářejí uživatelské rozhraní. V ASP.NET MVC Frameworku jsou kontroléry .NET třídy, které obsahují logiku potřebnou pro zpracování příchozího požadavku.“

Aby se třída stala kontrolérem, musím implementovat rozhraní `IController`. Jeho definice je ve výpisu 3-1. Rozhraní definuje pouze jednu metodu (`Execute`), která „je volána, když je (HTTP) požadavek nasměrován na daný kontrolér“ (Freeman, 2013).

Výpis 3-1: Definice rozhraní `IController`. *Zdroj: MVC Framework metadata*

```
public interface IController
{
    void Execute(RequestContext requestContext);
}
```

Implementací tohoto rozhraní lze vytvářet své vlastní kontroléry, jejich chování je možné téměř neomezeně přizpůsobovat vlastním potřebám. Vytvořit ale takto svůj vlastní typ kontroléru vyžaduje hodně práce. Pro programátora je proto v naprosté většině případů nejvhodnější vytvořit třídu, která dědí od třídy `Controller`. Třída `Controller` je součástí MVC Frameworku a kontroléry vytvořené přes nabídku Visual Studia (`Add Controller`) automaticky od této třídy dědí.

Díky třídě `Controller` je možné v kontroléru vytvářet tzv. akční metody, které zpracovávají příchozí požadavky. Esposito (2014) píše, že když má běhové prostředí ASP.NET MVC validní instanci vybrané třídy kontroléru, nasměruje samotné zpracování požadavku na objekt `ActionInvoker`. Objekt `ActionInvoker` se podle přijatého jména akční metody snaží tuto metodu v třídě kontroléru najít. Když ji najde, tak ji zavolá.

O to, který kontrolér bude vybrán a která jeho akční metoda bude zavolána, se stará *routing engine,* který toto určí podle URL příchozího požadavku. Akční metoda po jejím zavolání vrátí objekt typu `ActionResult`. Nejčastěji je vráceným objektem `ViewResult`

(který dědí od `ActionResult`), což je uživateli zobrazí HTML stránku v prohlížeči (Paz, 2013).

Akční metody obvykle komunikují s doménovou vrstvou, kdy získávají nebo mění doménový model. Také velmi často vytvářejí model pohledu, což je model obsahující pouze ta data, která pohled zobrazuje nebo je potřebuje ke správnému zobrazení. Tento model akční metoda pohledu předává.

Pohledům a směrování se budu detailněji věnovat dále.

3.4.2 Pohledy (Views)

Galloway a Haack (2014) píší, že pohledy jsou zodpovědné za poskytování uživatelského rozhraní uživateli. Na rozdíl od souborově zaměřených frameworků, jako jsou ASP.NET Web Forms nebo PHP, nejsou pohledy přístupné přímo. Webový prohlížeč nelze přímo nasměrovat na pohled a nechat si ho zobrazit. Místo toho je pohled vždy vytvořen kontrolérem, který mu také předá data k zobrazení.

Pokud kontrolér dědí od třídy `Controller`, může vytvářet pohledy pomocí metod zděděných právě od třídy `Controller`. Jsou to metody `View` a `PartialView`. Metoda `View` vrací zpět akční metodě objekt typu `ViewResult` a metoda `PartialView` vrací objekt typu `PartialViewResult`.

„Jedním z nejpoužívanějších typů, které vracejí akční metody ASP.NET MVC aplikace, je typ `ViewResult`. Když tento typ metoda vrací, znamená to, že MVC Framework má uživateli zobrazit určitý pohled (obvykle HTML obsah). Pohledy se ukládají do složky `Views` v aplikaci a často obsahují serverový kód, který HTML obsah pohledu vytvoří. Aby tento kód byl proveden, musí mít pohled specifikovaný *view engine*“ (Paz, 2013).

ASP.NET MVC 5 Framework obsahuje view engine jménem Razor. Je však také možné vytvořit svůj vlastní view engine, který bude z pohledů vytvářet HTML obsah, ale jak píše Freeman (2013), „téměř všechny MVC aplikace, které lidé vytvoří, si vystačí právě s enginem Razor. Jenom mizivá část projektů potřebuje svůj vlastní view engine“.

Díky Razor view engine mohu v pohledu vedle statického HTML kódu používat například C# kód. Tím lze upravovat obsah stránky, která se má odeslat prohlížeči. Pohledu budu často předávat nějaký model (nějaký .NET objekt) a data z tohoto modelu budu chtít

na stránce zobrazit. Například budu chtít v nějaké tabulce zobrazit data z kolekce, kterou pohledu předám. Razor mi umožní používat všechny konstrukce jazyka C# přímo v pohledu, mohu tedy předanou kolekci procházet pomocí `foreach` cyklu, nebo přistupovat k vlastnostem objektů kolekce.

3.4.3 Routing

Jak již bylo řečeno, aby byl požadavek zpracován správným a kontrolérem a správnou akční metodou, musí být na tento kontrolér a metodu nasměrován. To má na starosti *routing engine*.

Při startu aplikace se volá metoda `RegisterRoutes` třídy `RouteConfig`. Této metodě se předává (prázdná) kolekce cest `RouteCollection`. Uvnitř metody je možné vytvářet a mapovat nové cesty. „Cesta je nadefinována pomocí metody `MapRoute` třídy `RouteCollection`. Definice cesty obsahuje jako první parametr jméno cesty, jako druhý parametr vzor cesty a jako třetí parametr výchozí hodnoty pro jednotlivé elementy vzoru cesty“ (Paz, 2013). Ve výpisu 3-2 je zobrazeno volání metody `MapRoute`, které vytvoří Visual Studio při použití šablony projektu ASP.NET Web Application.

Výpis 3-2: Volání metody `MapRoute`. *Zdroj: šablona Visual Studia*

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new
    {
        controller = "Home",
        action = "Index",
        id = UrlParameter.Optional
    }
);
```

Ve druhém argumentu volání metody je definován tvar URL. Pomocí tohoto tvaru URL MVC Framework určí, na který kontrolér a kterou metodu požadavek směřuje. Pokud není v příchozí URL obsažena hodnota pro element `controller` nebo `action`, bude do nich dosazena hodnota z objektu předávaném jako třetí argument metody. Hodnota `id` je nastavena jako volitelná, pokud se tedy v URL nepředá, routing system provede nasměrování požadavku bez ní. Pokud je hodnota pro element `id` v URL specifikována, bude následně předána parametru akční metody `id`. O předání hodnoty z požadavku do parametru akční metody se stará funkce *model binding*, o které budu psát dále. V tabulce 3-1

je zobrazen přehled URL různých požadavků a u každé z nich je vypsán kontrolér a akční metoda, na kterou bude požadavek nasměrován.

Tabulka 3-1. Příklady nasměrování požadavku. *Zdroj: vlastní zpracování*

URL požadavku	Výsledek směrování
http://domena/	Kontrolér `Home`, akční metoda `Index`.
http://domena/Book	Kontrolér `Book`, akční metoda `Index`.
http://domena/Book/List	Kontrolér `Book`, akční metoda `List`.
http://domena/Book/Detail/4	Kontrolér `Book`, akční metoda `Detail`, předá 4 do parametru `id` metody `Book`.

Je samozřejmě možné přidávat další volání metod `MapRoute`, a definovat tak vlastní metody cesty, podle kterých bude MVC Framework určovat, na který kontrolér a akční metodu má požadavek nasměrovat, popřípadě jakou hodnotu má předat metodě v parametru.

Routing system má kromě funkce směrování požadavku na kontrolér a akční metodu ještě jednu funkci, a to je generování „výstupní URL“, tedy URL, která se po odeslání požadavku na server a vrácení HTML stránky akční metodou zobrazí v prohlížeči.

3.4.4 Model binding

„*Model binding* je proces vytvoření .NET objektu s použitím dat zaslaných prohlížečem v HTTP požadavku“ (Freeman, 2013).

Pokud bude URL příchozího požadavku ve tvaru `http://domena/Book/Detail/4` a akční metoda `Detail` bude přijímat parametr `id` typu `int`, model binding převede hodnotu `4` ze segmentu `{id}` na hodnotu typu `int` a vloží ji do parametru akční metody. Freeman (2013) také říká, že action invoker používá směrování cest ke zjištění metody, která má požadavek zpracovat, ale není schopen zavolat správnou metodu, pokud nemá pro parametr metody žádnou vhodnou hodnotu.

Pro MVC aplikaci je definován výchozí *model binder*, tedy objekt, který je zodpovědný právě za propojení hodnot z URL požadavku a parametru akční metody. Programátor si však může vytvořit vlastní model binder a přizpůsobit ho svým potřebám a potřebám aplikace.

Model binder je také schopen vkládat hodnoty do parametrů, které nejsou primitivního typu, ale typu komplexního. „Když je parametr akční metody komplexní typ (tedy jakýkoliv

typ, který nemůže být převeden pomocí třídy `TypeConverter`), třída `DefaulModelBinder` použije reflexi k získání sady veřejných vlastností objektu a na každou z nich potom naváže hodnotu“ (Freeman, 2013). Jednotlivé hodnoty, které se mají vložit do vlastností parametru komplexního typu, mohou být předány například pomocí formuláře.

3.4.5 Validace modelu

„Validace modelu je proces, kdy se aplikace ujišťuje, že obdržená data jsou vhodná pro navázání na vlastnosti modelu, a když tomu tak není, poskytne uživateli užitečné informace, které mu pomohou zjistit, kde je problém. První část tohoto procesu, kontrolující příchozí data, je jedním z klíčových způsobů, jak zachovat integritu doménového modelu. Odmítnutí dat, která nedávají v kontextu doménového modelu smysl, může zabránit nechtěným stavům aplikace. Druhá část, která pomáhá uživateli napravit problém, je neméně důležitá. Bez informací a zpětné vazby, kterou potřebuje k interakci s aplikací, by uživatel byl zmatený a frustrovaný“ (Freeman, 2013).

Validace modelu se provádí pomocí datových anotací (data annotations). Datové anotace jsou atributy definované ve jmenném prostoru `System.ComponentModel.DataAnnotations`. Uvedu zde některé atributy, které se používají nejčastěji. Atribut `[Required]` značí, že hodnota vlastnosti je vyžadována. Nesmí tedy být `null` a v případě, že je typu `String`, nesmí být hodnota ani prázdný řetězec. Pokud model binder při vkládání vstupních dat do vlastností modelu není schopen najít vhodnou hodnotu pro takto označenou vlastnost, uloží do vlastnosti `ModelState` kontroléru validační chybu. V akční metodě kontroléru je poté možné zkontrolovat, zda vlastnost `ModelState` obsahuje nějaké validační chyby, a podle toho se poté zachovat. Pokud například v `ModelState` vlastnosti nějaká validační chyba uložená bude, zobrazí akční metoda znovu stránku, kam se vstupní data zadávají, a u položek, jimž uživatel nezadal hodnotu, zobrazí zprávu o tom, že hodnota je vyžadována. V atributu lze také nastavit, jaká zpráva se uživateli zobrazí, pokud dojde k validační chybě. Ve výpisu 3-3 je zobrazena vlastnost s použitím atributu `[Required]` a nastavením validační zprávy.

Výpis 3-3: Použití datové anotace. *Zdroj: vlastní zpracování*

```
[Required(ErrorMessage = "Jméno je povinné.")]
public string Name { get; set; }
```

Dalšími atributy, které je možné použít, je například `[StringLength]`, jež umožňuje nastavit minimální a maximální délku řetězce, nebo `[Range]`, umožňující nastavit minimální a maximální číselnou hodnotu. Je také možné vytvářet svoje vlastní atributy.

3.4.6 Pomocné metody

Pomocné metody jsou metody, které je možné volat v pohledech a které většinou vytvoří v pohledu nějaký HTML obsah. Metody lze volat na různých vlastnostech pohledu, jako jsou `Html`, `Url` nebo `Ajax`. Vlastnosti `Html` a `Ajax` jsou definované ve třídě `WebViewPage` a vlastnost `Url` v její generické verzi.

Paz (2013) píše, že „myšlenkou na pozadí pomocných metod je možnost ukládání funkčností pro vytváření HTML obsahu do knihovny, a poté je používat napříč celou webovou aplikací.“

V ASP.NET MVC Frameworku je již předdefinováno mnoho pomocných metod, ale je samozřejmě možné vytvořit vlastní pomocné metody, které lze pak používat v pohledech. Pomocným metodám lze také v argumentech předávat různé hodnoty, které ovlivní konečnou podobu a vnitřní obsah vytvořeného HTML kódu. Často se pomocným metodám předávají hodnoty vlastností modelu pohledu. V tabulce 3-2 je zobrazeno několik základních pomocných metod definovaných v ASP.NET MVC Frameworku.

Tabulka 3-2: Některé základní pomocné metody. *Zdroj: vlastní zpracování*

`@Html.BeginForm()`	Vytvoří otvírací element formuláře
`@Html.ActionLink("Create new item", "Create")`	Vytvoří `<a>` element s hodnotou `href` atributu „`/{`*`kontrolér`*`}/Create`“ a vnitřní hodnotou „`Create new item`“.
`@Html.EditorFor(x => x.Name)`	Typ vytvořeného elementu bude určen na základě typu předané vlastnosti modelu. Pokud bude například vlastnost `Name` typu `String`, metoda vytvoří element `<input>` typu `text`.

3.5 Další frameworky a jazyky používané při vývoji MVC aplikací

3.5.1 jQuery a Ajax

Moderní webová aplikace se neobejde bez client-side skriptů. Pomocí těchto skriptů může být uživatelské rozhraní atraktivnější pro uživatele a může mu poskytovat užitečnou zpětnou vazbu. Skripty také mohou vylepšit výkon aplikace. Dvě nejpoužívanější javascriptové knihovny jsou jQuery a Ajax (Asynchronous JavaScript and XML). Díky knihovně Ajax je možné aktualizovat část stránky bez nutnosti načítat ji ze serveru celou.

„ASP.NET MVC implementuje koncept unobtrusive (nenápadného) JavaScriptu, což znamená, že v zájmu dodržení principu oddělení zodpovědností (separation of concerns) bude uchovávat JavaScript kód odděleně od HTML kódu. Unobtrusive JavaScript je založen na knihovně jQuery. jQuery poskytuje jednoduchý mechanismus k nalezení HTML elementů (jako jsou `<div>`, `<button>` nebo pole formuláře). Navíc jQuery obsahuje API pro manipulaci s těmito elementy“ (Pax, 2013). Freeman píše, že „MVC Framework obsahuje podporu pro *nenápadný* Ajax, což znamená, že je možné používat pomocné metody pro definování Ajax vlastností, než aby se kód přidával přímo do pohledů“.

3.5.2 Razor

„Razor je syntaxe šablon, která umožňuje plynule kombinovat kód s obsahem. I když zavádí několik nových klíčových slov, není to jazyk. Místo toho nechává programátora psát kód pomocí jazyků, které již zná“ (Chadwick, 2011).

Ve výpisu 3-4 je ukázka kódu pohledu vytvořeného pomocí Razor syntaxe. V pohledu se kombinuje obyčejný HTML kód s Razor výrazy. Všechny Razor výrazy musí začínat znakem `@`. Právě Razor výrazy mohu v pohledu volat metody a přistupovat k vlastnostem modelu.

Symbol `@` je také možné vkládat před bloky kódu, jak je vidět ve výpisu 3-4. „Všechny výrazy obsažené v (Razor) bloku jsou vyhodnoceny při zobrazení pohledu“ (Freeman, 2013).

Výpis 3-4: Ukázka použití Razor syntaxe. *Zdroj: vlastní zpracování*

```
@model Library.Domain.Entities.Book
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        <h3>@Model.Name</h3>
        -
        @Html.ActionLink("Delete book", "Delete")

    </div>
</body>
</html>
```

3.5.4 Unit testy

Každá profesionálně vyvíjená aplikace by měla být nějakou měrou a nějakým způsobem testována. Wilson (2014) píše, že „když mluvíme o testování software, odkazujeme se na celou škálu různých aktivit, včetně unit testování, akceptačního testování, agilního testování, testování výkonu a testování škálovatelnosti.“ V této kapitole se budu věnovat právě unit testování. Wilson (2014) uvádí čtyři hlavní vlastnosti, které by měly unit testy mít:

- **Testování malých částí produkčního kódu („units“)** – při psaní unit testu se programátor snaží testovat co nejmenší možnou funkcionalitu aplikace. V objektově orientovaném jazyku to bývá třída nebo častěji jedna metoda.
- **Testování v izolaci od ostatního produkčního kódu** – je nutné izolovat testovanou část kódu od ostatního kódu aplikace, se kterým testovaný kód komunikuje. Při zajištění takovéto izolace si programátor v případě neúspěšného testu může být jist, že chybu způsobuje právě testovaná část kódu. Ostatní části kódu by měly mít svoje vlastní unit testy.
- **Testování pouze veřejných koncových bodů** – unit test by neměl vědět příliš mnoho o vnitřním fungování třídy nebo metody, kterou testuje. Měl by se zaměřit pouze na testování jejich veřejných částí. Dodržením tohoto je snadnější měnit vnitřní implementaci tříd a metod, aniž by se rozbily unit testy.

- **Testování je prováděno automaticky s možnými výsledky „prošel“/„neprošel“ –** protože v projektu je obvykle velké množství unit testů, je žádoucí, aby provádění těchto testů bylo automatické a s jednoznačným výsledkem pro každý test, tedy „test prošel“/„test neprošel“. Pro automatizaci unit testů existuje mnoho frameworků. Jeden takový framework je dokonce integrován přímo do Visual Studia.

4. Vlastní práce

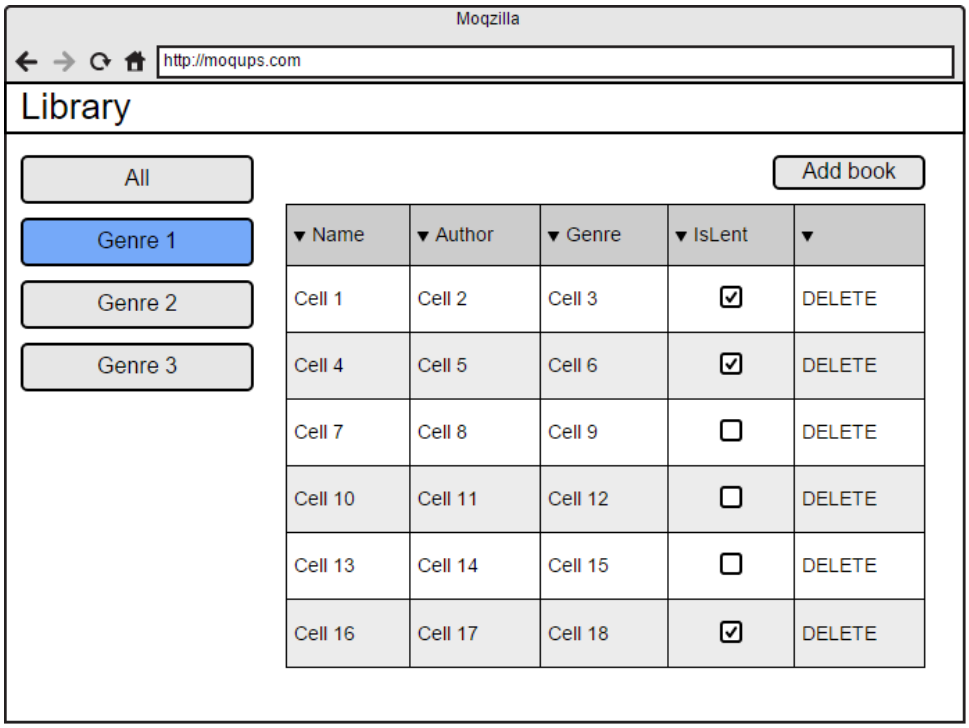
Technologie a frameworky, popsané v části 3 této práce budou, nyní aplikovány ve skutečné webové aplikaci. Bude popsán proces naplánování a vývoje jednoduché ASP.NET MVC aplikace. Účel aplikace bude poskytovat uživateli uživatelské rozhraní pro evidenci knih. Uživatel bude mít možnost pomocí aplikace do databáze přidávat nové tituly, zobrazovat a upravovat informace o těch již existujících, nebo je z databáze mazat. Také bude mít k dispozici jednoduchý filtr pro zobrazení knih vybraného žánru.

4.1 Návrh aplikace

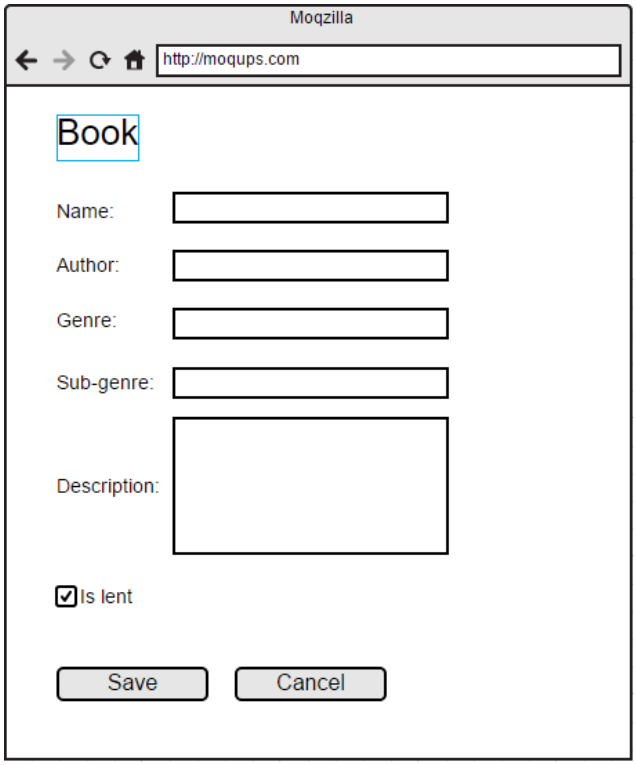
I když je navrhovaná aplikace velmi jednoduchá, bude rozdělena do dvou vrstev – na vrstvu prezentační a vrstvu doménovou. Pro každou z vrstev bude vytvořen samostatný projekt. Prezentační vrstva bude implementovat návrhový vzor *model-view-controller* pomocí MVC frameworku, avšak doménový model bude součástí vrstvy doménové. Ta bude současně zodpovědná za přístup k databázi. Pro komunikaci kontrolérů a doménového modelu bude implementován návrhový vzor *repository* s použitím *dependency injection*. Jako databáze bude použita vývojářská MS-SQL *localdb,* kterou je možno vytvořit a spravovat přímo ve Visual Studiu a není potřeba mít nainstalovaný SQL Server.

Grafické uživatelské rozhraní bude tvořeno hlavní stránkou a editační stránkou. Hlavní stránka bude obsahovat tabulku s výpisem knih a tlačítka s žánry, kterými bude možné filtrovat zobrazené knihy. Také na ní bude tlačítko pro přidání nové knihy do databáze. Na stránce bude možné kliknutím na název knihy zobrazit editační okno s detailními informacemi, nebo kliknutím na tlačítko *Delete* knihu z databáze vymazat. Na obrázku 4-1 je zobrazen wireframe hlavní stránky. Editační stránka bude zobrazovat detailní informace o knize, jako je název knihy, jméno autora, žánr, pod-žánr, popis knihy, zda je kniha půjčená a popřípadě další informace. Na stránce bude možné tyto vlastnosti knihy upravovat a ukládat, a také zadávat informace nově vytvořené knihy a novou knihu uložit. Na obrázku 4-2 je zobrazen wireframe editační stránky.

Obrázek 4-1: Wireframe hlavní stránky. *Zdroj: vlastní zpracování*[2]



Obrázek 4-2: Wireframe editační stránky. *Zdroj: vlastní zpracování*[2]



[2] Wireframe byl vytvořen ve webové aplikaci Moqups (http://moqups.com).

4.2 Příprava řešení

4.2.1 Založení řešení a projektů

Po spuštění Visual Studia vytvořím nové prázdné řešení a pojmenuji ho `MediaLibrary`. Uvnitř řešení následně vytvořím tři projekty. Pro první projekt vyberu v okně `Add New Project` možnost `ASP.NET Web Application` a pojmenuji ho `MediaLibrary.Presentation`. V následujícím okně vyberu pro projekt prázdnou šablonu (`Empty`) a zaškrtnu možnost `MVC` (to mi ve struktuře projektu vytvoří základní složky a reference). Tento projekt bude sloužit jako prezentační vrstva mojí aplikace. Právě pomocí MVC frameworku v něm bude implementováno uživatelské rozhraní a budou v něm vytvářeny kontroléry (controllers), pohledy (views) a pohledové modely (view models). Druhý projekt vytvořím jako obyčejnou knihovnu tříd (`Class Library`) a pojmenuji ho `MediaLibrary.Domain`. Toto bude doménová vrstva aplikace. To znamená, že v něm budou veškeré doménové modely (domain models) a veškerá business logika. V aplikaci bude tento projekt fungovat zároveň jako datová vrstva – bude zajišťovat přístup k databázi. Ve větších aplikacích bývají datová a doménová vrstva odděleny, ale z důvodů rozsahu této práce a očekávané malé velikost aplikace jsem se rozhodl tyto dvě vrstvy sloučit do jedné. Třetí a poslední projekt bude projekt testovací. Pojmenuji ho `MediaLibrary.UnitTests` a zvolím typ UnitTests. Tento projekt bude obsahovat unit testy pro testování funkčnosti prezentační a doménové (datové) vrstvy, tedy projektů `MediaLibrary.Presentation` a `MediaLibrary.Domain`.

Projekt `MediaLibrary.Presentation` nastavím jako spouštěcí (volba `Set as StartUp Project` z kontextového menu projektu). Aby si Visual Studio při spuštění debuggeru nevyžádalo URL pohledu, který zrovna edituji, ale vždy kořenovou URL aplikace, z kontextového menu projektu `MediaLibrary.Presentation` kliknu na volbu `Properties`, potom na záložku `Web` a v okně, které se mi zobrazí, zvolím možnost `Specific Page`.

4.2.2 Instalace NuGet balíčků a přidání odkazů mezi projekty

V mém řešení budu používat *dependency injection*, a také budu psát unit testy. Z toho důvodu pomocí příkazů uvedených ve výpisu 4-1 nainstaluji v panelu `Package Manager Console` Ninject framework (pro dependency injection) a Moq framework (pro usnadnění

psaní unit testů). Do projektů `MediaLibrary.Domain` a `MediaLibrary.UnitTesting` také nainstaluji balíček AspNet.Mvc.

Výpis 4-1: Příkazy pro instalaci NuGet balíčků. *Zdroj: vlastní zpracování*

```
Install-package Ninject -projectName MediaLibrary.Presentation
Install-package Ninject.Web.Common -projectName MediaLibrary.Presentation
Install-package Ninject.MVC5 -projectName MediaLibrary.Presentation
Install-package Ninject -projectName MediaLibrary.UnitTests
Install-package Ninject.Web.Common -projectName MediaLibrary.UnitTests
Install-package Ninject.MVC5 -projectName MediaLibrary.UnitTests
Install-package Moq -projectName MediaLibrary.Presentation
Install-package Moq -projectName MediaLibrary.UnitTests
Install-package Microsoft.Aspnet.Mvc -projectName MediaLibrary.Domain
Install-package Microsoft.Aspnet.Mvc -projectName MediaLibrary.UnitTests
```

Dále musím přidat závislosti mezi jednotlivé projekty. Projektu `MediaLibrary.Presentation` přidám odkaz (reference) na projekt `MediaLibrary.Domain` a do projektu `MediaLibrary.UnitTests` přidám odkazy na druhé dva, tedy `MediaLibrary.Presentation` i `MediaLibrary.Domain`. Navíc do testovacího projektu přidám odkazy na `System.Web` a `Microsoft.CSharp`. Projektu `MediaLibrary.Domain` přidám pouze odkaz na `System.ComponentModel.Data-Annotations`, ale už ne na žádný z ostatních projektů v mém řešení, protože doménová „bussiness“ vrstva (kterou tento projekt představuje) by neměla být závislá na vrstvě prezentační, a nejlépe ani na žádné jiné. Na projektu s testy by neměl být závislý žádný z projektů, které tvoří samotnou aplikaci.

4.2.3 Příprava DI resolveru a jeho registrace

Do projektu `MediaLibrary.Presentation` přidám složku `Infrastructure` a v ní vytvořím nový soubor `NinjectDependencyResolver.cs`. Obsah tohoto souboru je uveden ve výpisu 4-2.

Výpis 4-2: Obsah souboru NinjectDependencyResolver.cs. *Zdroj: vlastní zpracování*

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using Ninject;

namespace MediaLibrary.Presentation.Infrastructure
{
    public class NinjectDependencyResolver : IDependencyResolver
    {
        private IKernel kernel;

        public NinjectDependencyResolver (IKernel kernel)
        {
            this.kernel = kernel;
            AddBindings();
        }

        private void AddBindings()
        {
        }

        public object GetService(Type serviceType)
        {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType)
        {
            return kernel.GetAll(serviceType);
        }
    }
}
```

Třída `NinjectDependecyResolver` obsahuje metody `GetService` a `GetServices` které jsou předepsány rozhraním `IDependencyResolver` a budou volány MVC frameworkem vždy, když bude potřebovat konkrétní implementaci nějakého rozhraní. Ve třídě jsem si připravil soukromou metodu `AddBindings`, která se volá v konstruktoru třídy a párují se v ní jednotlivé třídy a rozhraní. Tělo metody zatím nechám prázdné, ale během programování aplikace sem budu jednotlivá párování tříd k rozhraním přidávat.

Nyní ještě musím zařídit, aby MVC Framework třídu `NinjectDependencyResolver` vůbec používal – musím ho v MVC frameworku zaregistrovat. Ve složce `App_Start` projektu `MediaLibrary.Presentation` je soubor `NinjectWebCommon.cs`, který zde byl vytvořen při instalaci Ninject balíčků. Ve třídě `NinjectWebCommons` v tomto souboru je definována metoda `RegisterServices`, která je automaticky volána vždy při startu aplikace (tělo metody je zatím prázdné). A právě v této metodě vytvořím instanci třídy

`NinjectDependencyResolver` (které v konstruktoru předám objekt typu `IKernel`) a jako parametr tuto instanci předám metodě `SetResolver` definované ve třídě `System.Web.Mvc.DependencyResolver`. Tímto jsem provedl registraci `Ninject-DependencyResolveru` v MVC frameworku. Celá metoda RegisterServices je ve výpisu 4-3.

Výpis 4-3: Registrace NinjectDependecyResolver v metodě RegisterServices. *Zdroj: vlastní zpracování*

```
private static void RegisterServices(IKernel kernel)
{
    DependencyResolver.SetResolver(
        new NinjectDependencyResolver(kernel));
}
```

4.3 Datová infrastruktura

4.3.1 Vytvoření doménového modelu a databáze

Nyní, když mám jednotlivé projekty připravené, je potřeba vytvořit doménový model a databázi. Účel aplikace je evidence knih, jejich jednotlivých kopií, jejich dostupnosti atd. Proto tou nejdůležitější entitou, se kterou bude aplikace pracovat, je kniha. V projektu `Library.Domain` vytvořím složku `Entities` a v ní třídu, kterou pojmenuji `Book`. Výpis 4-4 zobrazuje obsah souboru `Book.cs`.

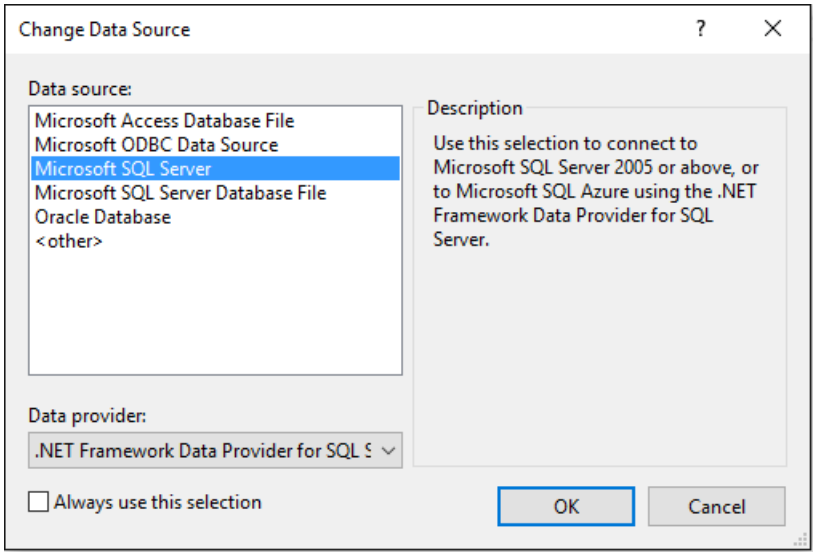
Informace o knihách musí být někde uloženy, založím tedy databázi. Pro vytvoření mé databáze jsem zvolil Microsoft SQL Server, a to především z důvodu, že Visual Studio mi umožňuje vytvořit zjednodušenou SQL Server databázi (LocalDB), která je ale pro vývoj zcela postačující. LocalDB se instaluje automaticky spolu s Visual Studiem, nemusím proto instalovat a nastavovat samostatnou instanci SQL Serveru a moji databázi mohu také spravovat přímo ve Visual Studiu.

Výpis 4-4: Obsah souboru `Book.cs`. *Zdroj: vlastní zpracování*

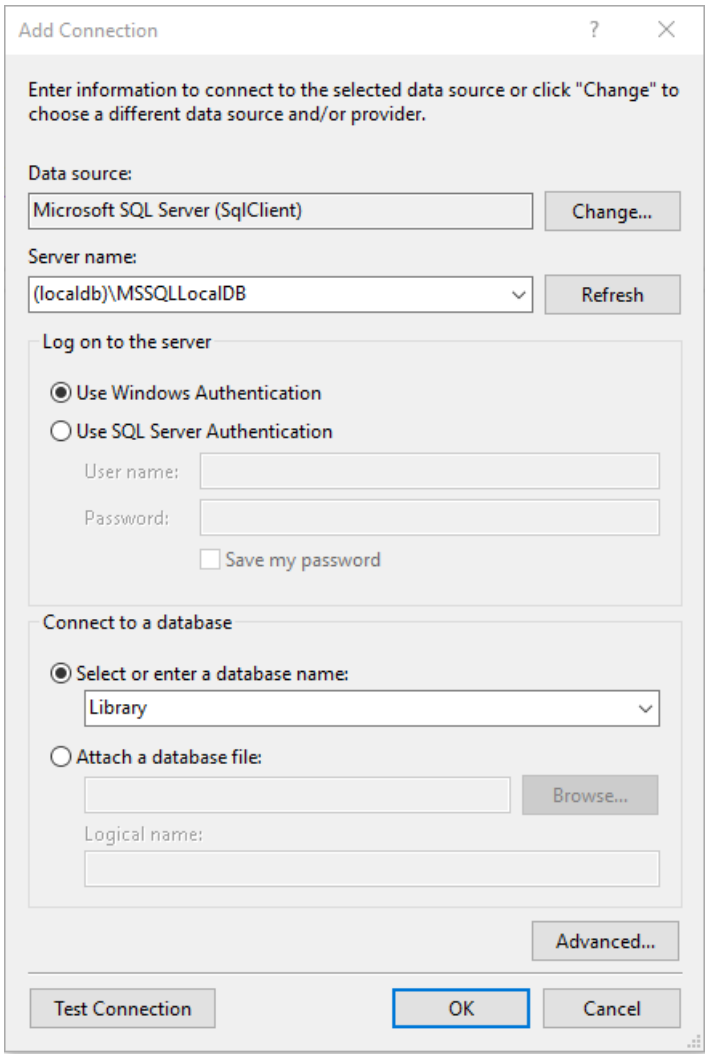
```
namespace Library.Domain
{
    public class Book
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public string Author { get; set; }
        public string Genre { get; set; }
        public string Description { get; set; }
        public bool IsLent { get; set; }
    }
}
```

V okně `Server Explorer` kliknu na tlačítko `Connect to Database` a zobrazí se okno `Add Connection`. V něm kliknu na tlačítko `Change` a v okně, které se mi otevře (obrázek 4-3), vyberu jako datový zdroj (`Data source`) položku `Microsoft SQL Server` a jako poskytovatele (`Data provider`) zvolím `.NET Framework Data Provider for SQL Server`. Potvrdím tlačítkem `OK` a v okně `Add Connection` poté do pole `Server name` napíšu `(localdb)\MSSQLLocalDB`, což je název mého lokálního SQL Serveru. Do pole `Select or enter a database name` napíšu název mé budoucí databáze. Pojmenuji ji stejně jako aplikaci, tedy `Library`. Na obrázku 4-4 je vidět nastavení v okně `Add Connection`. Tlačítkem `OK` vytvořím připojení. Visual Studio se mně následně zeptá, jestli chci vytvořit novou databázi. Kliknu na tlačítko `Yes` a nechám databázi vytvořit.

Obrázek 4-3: Okno výběru datového zdroje. *Zdroj: Visual Studio*



Obrázek 4-4: Okno přidání připojení k databázi. *Zdroj: Visual Studio*



Nově vytvořená databáze `Library` se mi zobrazí jako nová položka v okně `Server Explorer` pod `Data Connections`. Rozbalím ji, pravým tlačítkem kliknu na složku `Tables` a z kontextového menu zvolím možnost `Add New Table`. V okně, které se mi zobrazí, mohu nyní vytvořit tabulku pro entitu Book. Je možné ji vytvořit v podokně `Design` nebo pomocí skriptu v podokně `T-SQL`. Já ji vytvořím SQL skriptem. Celý skript pro založení tabulky `Books` (tabulky pojmenovávám v množném čísle) je zobrazen ve výpisu 4-5. Po napsání ho spustím tlačítkem `Update` a v okně, které se mi následně zobrazí, potvrdím volbu pomocí tlačítka `Update Database`. Ve sloupečcích tabulky se budou ukládat jednotlivé informace o knihách, jako je jejich název, jméno autora, žánr atd.

Výpis 4-5: T-SQL skript pro vytvoření tabulky `Books`. *Zdroj: vlastní zpracování*

```
CREATE TABLE [dbo].[Books] (
    [ID] INT NOT NULL PRIMARY KEY IDENTITY,
    [Name] NVARCHAR(50) NOT NULL,
    [Author] NVARCHAR(50) NOT NULL,
    [Genre] NVARCHAR(20) NOT NULL,
    [Subgenre] NVARCHAR(20) NOT NULL,
    [Description] NVARCHAR(MAX) NULL,
    [IsLent] BIT NOT NULL
);
```

Aby se mi aplikace lépe testovala, naplním tabulku `Books` fiktivními daty (knihami). V okně `Server Explorer` kliknu pravým tlačítkem na tabulku `Books` a z kontextového menu vyberu možnost `Show Table Data`. Zobrazí se mi okno, ve kterém mohu přidávat nebo upravovat položky tabulky. Na obrázku 4-5 je zobrazeny řádky s daty, které do tabulky vložím. Do sloupečku `ID` žádné hodnoty nezadávám, protože jsem při vytváření tabulky tento sloupec označil atributem `IDENTITY`, a proto SQL Server každému novému řádku vygeneruje automaticky nové, unikátní ID (ve výchozím nastavení se ID vytváří od 1 a po jedné se inkrementuje).

Obrázek 4-5: Fiktivní data tabulky `Books`. *Zdroj: vlastní zpracování*

ID	Name	Author	Genre	Subgenre	Description	IsLent
2	Pillars of the Ea...	Ken Follett	Novel	Historical	Tells the story o...	False
3	Fall of Giants	Ken Follett	Novel	Historical	This is a huge n...	False
4	Winter of the W...	Ken Follett	Novel	Historical	Berlin in 1933 is...	False
6	Edge of Eternity	Ken Follett	Novel	Historical	*NULL*	False
7	Neuromancer	William Gibson	Novel	Sci-fi	Cyber-punk ma...	False
9	Gone girl	Gillian Flynn	Novel	Thriller	*NULL*	False
10	HTML & CSS : ...	Jon Duckett	Education	Computing	*NULL*	True
18	Clean Code	Robert C. Martin	Education	Computing	Even bad code ...	False
21	Design Patterns	Erich Gamma	Education	Computing	*NULL*	False
24	The Martian	Andy Weir	Novel	Sci-fi	*NULL*	False
25	Nineteen Eight...	George Orwell	Novel	Drama	*NULL*	False
26	A Game of Thr...	George R. R. M...	Novel	Fantasy	The first volum...	False
28	The Hobbit : or ...	J. R. R. Tolkien	Novel	Fantasy	*NULL*	True
29	The Ultimate Hi...	Douglas Adams	Novel	Sci-fi	*NULL*	True
32	The Silmarillion	J. R. R. Tolkien	Myths and lege...	Fantasy	Designed to tak...	False
35	Hyperion	Dan Simmons	Novel	Sci-fi	On the world c...	False
NULL	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*

4.3.2 Propojení doménového modelu s databází

Model i databázi mám nyní připravené, ale musím je mezi sebou nějak propojit, abych mohl hodnoty modelu do databáze ukládat a zase je z databáze do modelu načítat. Toho mohu docílit buď využitím klasického ADO.NET frameworku a jeho standardních tříd, jako

je SqlCommand, SqlDataReader aj., nebo pomocí nějakého ORM (object-relational mapper) frameworku. Já použiji ORM, a to konkrétně Entitity Framework od Microsoftu[3]. Entity Framework pro mne bude zajišťovat přenos dat mezi databází a modelem na nízké úrovni. Nebudu tak muset ručně psát jednotlivé SQL dotazy, mapovat model na databázové entity atd. Ve výpisu 4-6 jsou dva příkazy, pomocí kterých nainstaluji NuGet balíčky Entity Framework do projektů `Library.Domain` a `Library.Presentation`.

Výpis 4-6: Příkazy pro instalaci NuGet balíčku EntityFramework. *Zdroj: vlastní zpracování*

```
Install-package EntityFramework -projectName Library.Domain
Install-package EntityFramework -projectName Library.Presentation
```

Nyní, když mám Entity Framework nainstalovaný, nastavím *connection string*, aby Entity Framework věděl, ke které databázi se má připojit a také jak. Element `<connectionStrings>`, uvedený ve výpisu 4-7, vložím do kořenového elementu `<configuration>` v souboru `Web.config`, který se nachází v projektu `Library.Presentation`. Ve výpisu jsem musel hodnotu atributu `connectionString` rozdělit na více řádků, aby se vešel na stránku, ale v kódu musí být řetězec celý na jednom řádku.

Výpis 4-7: Element `<connectionStrings>`. *Zdroj: vlastní zpracování*

```
<connectionStrings>
  <add name="EFDbContext"
       connectionString="Data Source=(localdb)\MSSQLLocalDB;
                         Initial Catalog=Library;
                         Integrated Security=True"
       providerName="System.Data.SqlClient" />
</connectionStrings>
```

Dalším krokem je vytvoření třídy, která bude zajišťovat samotné propojení databáze s modelem. Pro tuto třídu v projektu `Library.Domain` vytvořím novou složku, kterou pojmenuji `Concrete`. Třídu pojmenuji `EFDbContext`. Třída `EFDbContext` dědí od třídy `System.Data.Entity.DbContext`, což mi zajistí automatické mapování modelu na entitu databáze. Já už jenom definuji vlastnost, která bude určovat spojení konkrétního modelu s konkrétní databázovou entitou. Název vlastnosti odpovídá názvu databázové

[3] „Entity Framework je ve skutečnosti technologické rozšíření Microsoft ActiveX Data Object (ADO.NET)“ (Mueller, 2013)

tabulky, tedy `Books`, a typový parametr vlastnosti odpovídá názvu modelu, tedy `Book`. Obsah souboru `EFDbContext.cs` je uveden ve výpisu 4-8.

Výpis 4-8: Obsah souboru EFDbContext.cs. *Zdroj: vlastní zpracování podle Freemana (2013)*

```
using System.Data.Entity;
using Library.Domain.Entities;

namespace Library.Domain.Concrete
{
    public class EFDbContext : DbContext
    {
        public DbSet<Book> Books { get; set; }
    }
}
```

4.3.3 Získávání dat z databáze

Potřebuji nějaký vhodný způsob, jak data uložená v databázi načítat do běžící aplikace. Bylo by vhodné, abych přitom udržel odděleně doménový model a kód pro získávání dat. K tomuto účelu je vhodné použít návrhový vzor Repository.

V projektu `Library-Domain` vytvořím novou složku pojmenovanou `Abstract` a v ní vytvořím rozhraní (interface) `IBookRepository`. Obsah souboru `IBookRepository.cs` je ve výpisu 4-9.

Výpis 4-9: Obsah souboru IBookRepository.cs. *Zdroj: vlastní zpracování*

```
using Library.Domain.Entities;
using System.Collections.Generic;

namespace Library.Domain.Abstract
{
    interface IBookRepository
    {
        IEnumerable<Book> Books { get; }
    }
}
```

Třída, implementující toto rozhraní, musí definovat vlastnost `Books`, která je typu `IEnumerable<Book>`. Rozhraní umožňuje nějakému (volajícímu) objektu získat kolekci objektů typu `Book`. Volající objekt neví (a ani to vědět nepotřebuje) kdo, jak a kde tyto objekty typu `Book` získá. A právě toto je princip návrhového vzoru `Repository`.

Jako další vytvořím konkrétní repositář pro objekty typu `Book`. Ve složce `Concrete` projektu `Library.Domain` vytvořím třídu a pojmenuji ji `EFBookRepository`. Třída bude implementovat již vytvořené rozhraní `IBookRepository` a bude používat objekt typu `EFDbContext` pro získávání dat z databáze. Obsah souboru `EFBookRepository.cs` je ve výpisu 4-10.

Výpis 4-10: Obsah souboru EFBookRepository.cs. *Zdroj: vlastní zpracování*

```
using Library.Domain.Abstract;
using System.Collections.Generic;
using Library.Domain.Entities;

namespace Library.Domain.Concrete
{
    public class EFBookRepository : IBookRepository
    {
        EFDbContext context = new EFDbContext();

        public IEnumerable<Book> Books
        {
            get { return context.Books; }
        }
    }
}
```

4.4 Základní prezentační vrstva podle vzoru MVC

4.4.1 První kontrolér a pohled

Mám připravenou základní datovou infrastrukturu pro získávání dat z tabulky `Books` v databázi. Bylo by dobré tedy vytvořit kontrolér a k němu nějaký jednoduchý pohled a vyzkoušet, jak načítání dat z databáze funguje.

V okně `Solution Explorer` kliknu pravým tlačítkem na složku `Controllers` projektu `Library.Presentation` a v kontextovém menu, které se mi zobrazí, vyberu možnost `Add` ➔ `Controller`. V okně `Add Scaffold`, které se mi otevře, vyberu možnost `MVC 5 controller – Empty` a volbu potvrdím tlačítkem `Add`. Následně v okně `Add Controller` nový kontrolér pojmenuji jako `BookController` a kliknu na `Add`. Obsah souboru `BookController.cs`, který byl vytvořen, upravím tak, jak je uvedeno ve výpisu 4-11.

Výpis 4-11: Obsah souboru `BookController.cs`. *Zdroj: vlastní zpracování*

```
using Library.Domain.Abstract;
using System.Web.Mvc;

namespace Library.Presentation.Controllers
{
    public class BookController : Controller
    {
        private IBookRepository repository;

        public BookController(IBookRepository repository)
        {
            this.repository = repository;
        }

        public ViewResult List()
        {
            return View(repository.Books);
        }
    }
}
```

Objekt (kontrolér) `BookController` dědí od objektu `Controller`, který poskytuje množství metod pro zpracovávání příchozích HTTP dotazů. Ve třídě jsem vytvořil privátní pole typu `IBookRepository` a konstruktor přijímá parametr, který je také typu `IBookRepository`. To znamená, že kontrolér není závislý na žádné konkrétní implementaci repositáře, ale pouze na rozhraní `IBookRepository`, což je přesně to, čeho jsem chtěl dosáhnout. Já už jen musím zařídit, aby se konstruktoru předala konkrétní implementace repositáře (v mém případě `EFBookRepository`). Toho docílím vložením závislosti pomocí `Ninject Frameworku`, který již mám připravený. V souboru `NinjectDependencyResolver.cs` projektu `Library.Presentation` mám připravenou soukromou metodu `AddBindings`, která je zatím prázdná. Nyní do ní přidám kus kódu, který „spáruje“ rozhraní `IBookRepository` s třídou `EFBookRepository`. Celá metoda `AddBindings` je vidět ve výpisu 4-12.

Výpis 4-12: Metoda `AddBindings`. *Zdroj: vlastní zpracování*

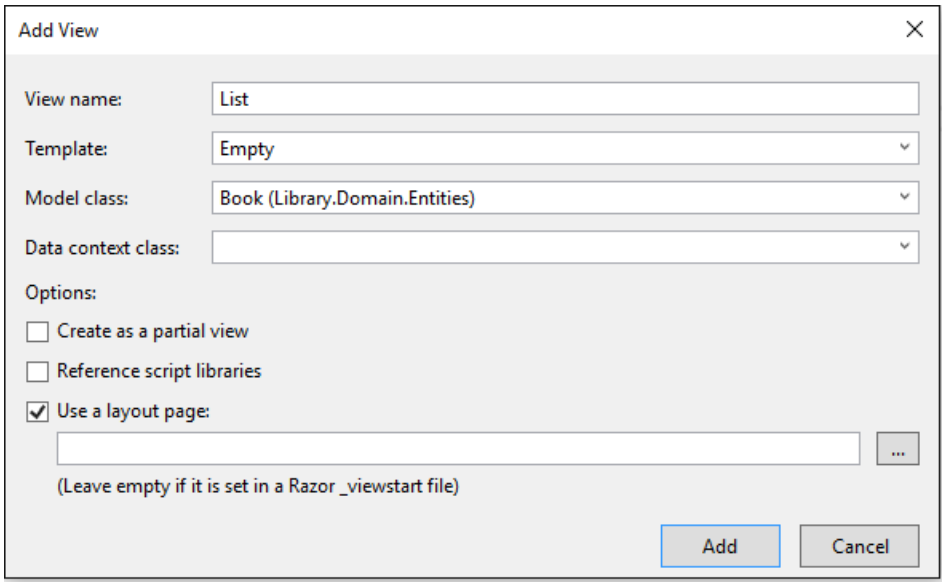
```
private void AddBindings()
{
    kernel.Bind<IBookRepository>().To<EFBookRepository>();
}
```

Od tohoto okamžiku vždy, když bude vytvořena instance třídy `BookController`, Ninject Framework vloží do konstruktoru místo parametru typu `IBookRepository` objekt typu `EFBookRepository`.

V kontroléru jsem vytvořil metodu pojmenovanou `List`. Toto je tzv. *akční metoda*, (action method) která při zavolání vytvoří pohled vypisující všechny položky, které získá z repositáře. Akční metoda sama volá metodu `View`, zděděnou od třídy `Controller`, a předá jí jeden parametr (`repository.Books`). Jelikož jsem metodě `View` nepředal v parametru název konkrétního pohledu, metoda vytvoří a vrátí výchozí pohled přiřazený k akční metodě `List`. Předávaným parametrem `repository.Books`, který je typu `IEnumerable<Library.Entities.Book>`, je naplněna vlastnost `Model` tohoto pohledu. Akční metoda vrací pohled vytvořený metodou `View` jako objekt typu `ViewResult`.

Aby mohla akční metoda vůbec nějaký pohled vrátit, musím ho nejdříve vytvořit v kódu. Nejjednodušší způsob je kliknutím pravým tlačítkem kdekoliv v těle metody `List` třídy `BookController` a v kontextovém menu vybrat možnost `AddView`. Zobrazí se okno `Add View`, které vyplním tak, jak zobrazeno na obrázku 4-6.

Obrázek 4-6: Okno pro přidání pohledu `Book/List.cshtml`. *Zdroj: Visual Studio*



Pohled jsem pojmenoval `View` (tedy stejně jako akční metodu, která tento pohled bude vracet) a nepoužil jsem pro něj žádnou šablonu. Pouze jsem mu nastavil model `Book`, čímž

jsem ho vytvořil jako silně typový. Jako `Data Context Class` bych normálně vybral třídu `Library.Domain.Concrete.EFDbContext` a Visual Studio by v této třídě vytvořilo vlastnost `DBSet<Book> Books`, ale já jsem tuto vlastnost již vytvořil dříve ručně, proto nechám pole prázdné. Zaškrtnutím možnosti `Use a Layout page` řeknu Visual Studiu, aby pro vytvořený pohled použil tzv. Layout stránku. Pokud do pole níže nevyberu žádnou již existující Layout stránku, Visual Studio vytvoří novou, pojmenovanou `_Layout.cshtml`, ve složce `Library.Domain/Views/Shared`, a také ve složce `Library.Domain/Views` vytvoří soubor `_ViewStart.cshtml`. V souboru `_ViewStart.cshtml` automaticky nastaví, že pohledy mají jako svoji Layout stránku používat právě `_Layout.cshtml`. Při vytvoření `_Layout.cshtml` souboru Visual Studio automaticky vytvoří jeho obsah. Pro potřeby mé aplikace ale upravím Layout stránku do podoby zobrazené ve výpisu 4-13.

Výpis 4-13: Obsah souboru `_Layout.cshtml`. *Zdroj: vlastní zpracování*

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Vždy, když akční metoda `List` kontroléru `BookController` bude vytvářet pohled `List`, tak tento pohled vloží na místo volání metody `RenderBody` a celou Layout stránku i s vytvořeným pohledem vrátí. Nyní ještě upravím obsah souboru `List.cshtml`, jak je zobrazeno ve výpisu 4-14. I když jsem pohledu při jeho vytváření nastavil model jako typ `Library.Domain.Entities.Book`, mohu mu předat i generickou kolekci tohoto typu (což jsem také udělal, když jsem metodě `View`, kterou volá akční metoda List kontroléru `BookController`, předal jako parametr objekt `repository.Books`, který je typu `IEnumerable<Book>`). V cyklu `foreach` vytvořím pro každý jednotlivý objekt (knihu) kolekce `<div>` element, ve kterém se budou vypisovat základní informace o knize.

Výpis 4-14: Obsah souboru List.cshtml. *Zdroj: vlastní zpracování*

```
@using Library.Domain.Entities
@model IEnumerable<Book>

@{
    ViewBag.Title = "Books";
}

@foreach (Book book in Model)
{
    <div>
        <h3>@book.Name</h3>
        @book.Author<br />
        @book.Genre - @book.Subgenre<br />
    </div>
}
```

4.4.2 Úprava směrování URL

Chci, aby aplikace při spuštění automaticky „nasměrovala“ požadavek na akční metodu `List` kontroléru `BookController`, která vrátí výchozí pohled `List`. Toho docílím malou úpravou metody `RegisterRoutes` v souboru `Library.Presentation/App_Start/RouteConfig.cs`. Metoda po úpravě je zobrazena ve výpisu 4-15.

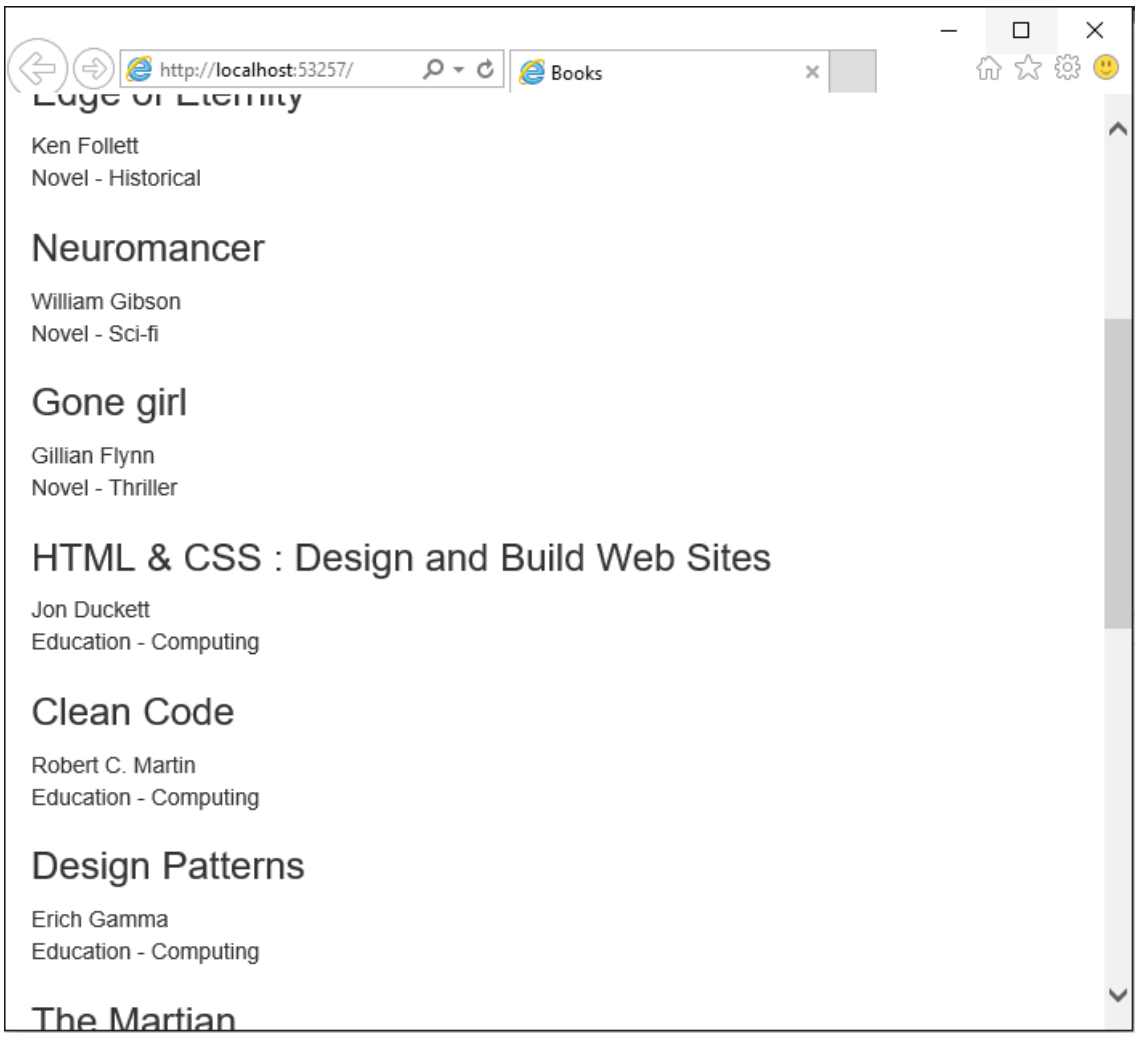
Výpis 4-15: Upravená metoda `RouteCongic.RegisterRoutes`. *Zdroj: vlastní zpracování*

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new {
            controller = "Book",
            action = "List",
            id = UrlParameter.Optional}
    );
}
```

Jediné, co jsem změnil, byly hodnoty přiřazované proměnným `controller` a `action` objektu `defaults`. Hodnotu `Home` jsem změnil za `Book` a hodnotu `Index` jsem změnil za `List`. Hodnoty `controller` a `action` říkají, na jaký kontrolér a akční metodu se má přicházející HTTP požadavek přesměrovat, pokud tímto požadavkem nebyly tyto hodnoty předané v URL.

Pokud bych nyní aplikaci spustil, zobrazí mi jednoduchý výpis dat z dat z databáze. Prohlížeč si od serveru vyžádá výchozí URL (v mém případě http://localhost:53257/). Jelikož tato URL nespecifikuje název kontroléru ani akční metody, směrovací systém automaticky požadavek nasměruje na kontrolér `BookController` a jeho akční metodu `List`. Tato akční metoda zavolá metodu `View`, kterou `BookController` zdědil od třídy `Controller` a předá jí kolekci objektů typu `Book` získanou z repositáře `EFBookRepository`, jehož instance byla kontroléru předána při vytváření. Metoda `View` vytvoří s použitím kolekce objektů `Book` pohled `List`, který je následně vložen do Layout stránky. Layout stránka i s vloženým pohledem je nakonec jako obyčejná HTML stránka vrácena ze serveru prohlížeči, který ji zobrazí uživateli. Na obrázku 4-7 je část zobrazené HTML stránky.

Obrázek 4-7: Část stránky zobrazené v prohlížeči. *Zdroj: vlastní zpracování*



4.4.3 Stylizace stránky pomocí Bootstrap knihovny

Data z databáze se mi sice v prohlížeči zobrazují, ale po prezentační stránce na tom aplikace není zrovna nejlépe. Budu se proto nyní chvíli věnovat stylizaci zobrazovaného obsahu. K tomu použiji samozřejmě kaskádové styly (CSS), konkrétně ty definované v Bootstrap knihovně. Jelikož jsem si při zakládání projektu `Library.Presentation` zvolil prázdnou šablonu a ne šablonu `MVC`, musím si nainstalovat NuGet balíček Bootstrap, který obsahuje všechny potřebné CSS knihovny. Ve výpisu 4-16 je zobrazen příkaz k nainstalování balíčku.

Výpis 4-16: Příkaz k nainstalování Boostrap NuGet balíčku. *Zdroj: vlastní zpracování*

```
Install-Package Bootstrap -projectName Library.Presentation
```

Po nainstalování balíčku upravím soubor `_Layout.cshtml`. Do elementu `<head>` přidám odkazy (linky) na CSS knihovny, které se mi při instalaci NuGet balíčku nainstalovaly do složky `Content`. Také upravím obsah elementu `<body>`. Rozdělím tím stránku na tři části. V horní části bude hlavička stránky, která bude přes celou její šíři. Obsah stránky pod hlavičkou bude rozdělen na dva sloupce. Levý sloupec bude sloužit jako navigační panel a pravý bude zobrazovat jednotlivé položky (knihy). Obsah souboru `_Layout.cshtml` po úpravě je vidět ve výpisu 4-17.

Výpis 4-17: Obsah souboru `_Layout.cshtml`. *Zdroj: vlastní zpracování*

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.min.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div class="navbar navbar-default" role="navigation">
        <a class="navbar-brand" href="#">Library</a>
    </div>
    <div class="container-fluid">
        <div class="row panel">
            <div id="genres" class="col-xs-4">
            </div>
            <div class="col-xs-8">
                @RenderBody()
            </div>
        </div>
    </div>
</body>
</html>
```

Stránku jsem rozdělil na dva sloupce a jejich šířku jsem nastavil v poměru 1 : 2 pomocí `class` atributů s hodnotami `col-xs-4` a `col-xs-8`, pro které jsou definovány CSS styly právě v knihovně Boostrap. Aby to ale fungovalo, musím oba `<div>` elementy s těmito atributy obalit jiným `<div>` elementem, kterému nastavím hodnotu `class` atributu jako `row panel`, kde právě hodnota `row` způsobuje možnost rozdělovat oddíl na sloupce. Počet sloupců v oddílu je vždy 12. Podřízené oddíly by měly vždy dohromady zabírat přesně 12 sloupců. Počet sloupců, které oddíl zabírá, je určen číslem v hodnotě atributu `class`. První oddíl (navigační panel) tedy bude zabírat 4 sloupce a ten druhý (oddíl s výpisem položek) sloupců 8. Dvě písmena `xs` v hodnotě atributu určují, že toto rozdělení na sloupce bude aktivní i pro ta nejnižší rozlišení obrazovek (tj. s šířkou do 768 pixelů).

Provedl jsem základní stylizaci Layout stránky, ale musím ještě upravit pohled `List`, tedy to, jak budou vypadat jednotlivá zobrazení položky s informacemi o knížkách. Soubor `List.cshtml`, jak je zobrazeno ve výpisu 4-18.

Výpis 4-18: Upravený soubor `List.cshtml`. *Zdroj: vlastní zpracování*

```
@using Library.Domain.Entities
@model IEnumerable<Book>

@{
    ViewBag.Title = "Books";
}

<table class="table table-striped table-bordered">
    <thead>
        <tr>
            <td><strong>Name</strong></td>
            <td><strong>Author</strong></td>
            <td><strong>Genre</strong></td>
            <td><strong>Subgenre</strong></td>
            <td><strong>Is lent?</strong></td>
        </tr>
    </thead>
    <tbody>
        @foreach (Book book in Model)
        {
            <tr>
                <td>@book.Name</td>
                <td>@book.Author</td>
                <td>@book.Genre</td>
                <td>@book.Subgenre</td>
                <td>@book.IsLent</td>
            </tr>
        }
    </tbody>
</table>
```

Aby byl výpis položek přehlednější, použil jsem tabulku, kterou jsem stylizoval opět pomocí Bootstrap stylů. V tabulce jsem vytvořil nadpisy sloupců podle názvů sloupečků tabulky `Books` v databázi a pro každou jednu knihu, která se z databáze získá, vytvořím v tabulce pomocí cyklu `foreach` jeden řádek.

4.4.4 Filtrování podle žánru

Pomocí Bootstrap stylů jsem si stránku rozdělil na dva sloupce. V tom pravém se zobrazují v tabulce vypsaná data z databáze. V tom levém se zatím nezobrazuje nic. Já chci, aby se mi zde zobrazovaly žánry knih, podle kterých budu moci filtrovat položky v tabulce. Začnu vytvořením nového kontroléru, který bude zajišťovat vytvoření pohledu se seznamem žánrů. Kontrolér vytvořím ve složce `Library.Presentation/Controllers` a pojmenuji ho `NavigationController`. Obsah souboru upravím, jak je zobrazeno ve výpisu 4-19.

Výpis 4-19: Obsah souboru NavigationController.cs. *Zdroj: vlastní zpracování*

```
using Library.Domain.Abstract;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;

namespace Library.Presentation.Controllers
{
    public class NavigationController : Controller
    {
        private IBookRepository repository;

        public NavigationController(IBookRepository repository)
        {
            this.repository = repository;
        }

        public PartialViewResult GenresMenu()
        {
            IEnumerable<string> genres =
                repository.Books
                    .Select(b => b.Genre)
                    .Distinct()
                    .OrderBy(g => g);
            return PartialView(genres);
        }
    }
}
```

Kontrolér v konstruktoru obdrží instanci repositáře `EFBookRepository`. Stejně jako kontroléru `BookController` mu tuto instanci do parametru vloží Ninject Framework. V kontroléru jsem smazal akční metodu Index, kterou vytvořilo Visual Studio a místo ní jsem vytvořil akční metodu `GenresMenu`. Metoda ze všech objektů `Book` v repositáři získá unikátní hodnoty vlastnosti `Genre`, seřadí je podle abecedy a předá metodě `PartialView`, která vytvoří a vrátí *dílčí pohled* (partial view). Při volání metody `PartialView` jsem neuvedl jméno pohledu, který má metoda vytvořit, a proto vytvoří pohled výchozí pro akční metodu `GenresMenu`. Nyní tento pohled vytvořím v kódu. Pravým tlačítkem kliknu na složku Library.Presentation/Views/Navigation a z kontextového menu vyberu Add ➔ `MVC 5 Partial View (Razor)`. V okně, které se mi otevře, pojmenuji nový pohled jako `GenresMenu` a kliknu na `OK`. Obsah nově vytvořeného souboru `GenresMenu.cshtml` upravím do podoby zobrazené ve výpisu 4-20.

Výpis 4-20: Dílčí pohled `GenresMenu`. *Zdroj: vlastní zpracování*

```
@model IEnumerable<string>

@Html.ActionLink("All", "List", "Book", null,
    new { @class = "btn btn-block btn-default" })

@foreach (string link in Model)
{
    @Html.RouteLink(link,
    new {
        controller = "Book",
        action = "List",
        genre = link
    },
    new { @class = "btn btn-block btn-default" })
}
```

V pohledu bude jako první vytvořen pomocnou metodou `ActionLink` element `<a>` s odkazem na akční metodu `List` kontroléru `Book`. V odkazu nebude žádná hodnota pro parametr `genre` akční metody, proto se položky z repositáře nebudou podle žádné hodnoty filtrovat a v tabulce se zobrazí úplně všechny. Dále se v pohledu vytvoří odkazy pro každý jeden z žánrů modelu pohledu. Pomocná metoda `RouteLink` funguje stejně jako metoda `ActionLink`, ale mohu jí předat objekt, ve kterém definuji podobu URL, která se vytvoří a vloží do odkazu. Oběma metodám předám také objekt, ve kterém definuji atribut `class`[4]. Hodnoty, které mu předám, jsou třídy, pro které jsou definované CSS styly v Bootstrap knihovně. Tímto bude každý odkaz vypadat jako tlačítko a budou seřazeny vertikálně.

Nyní musím zařídit, aby se mi pohled se žánry na stránce vůbec zobrazoval. Toho docílím voláním pomocné metody `Html.Action` z Layout stránky. V souboru `_Layout.cshtml` již mám připravený sloupec, ve kterém pohled bude. Je to element `<div>`, kterému jsem definoval atribut `class="col-xs-4"`. Ve výpisu 4-21 je tento element i s voláním metody `Html.Action`.

[4] „Pokud je potřeba použít identifikátor, který koliduje s klíčovým slovem jazyka, je možno to udělat vložením znaku @ před identifikátor. Znak @ nebude součástí identifikátoru“ (Albahari, Albahari, 2012).

Výpis 4-21: Upravený element s voláním metody `Html.Action`. *Zdroj: vlastní zpracování*

```
<div id="genres" class="col-xs-4">
    @Html.Action("GenresMenu", "Navigation")
</div>
```

Metoda `Html.Action` zavolá tzv. dětskou akční metodu, což je v tomto případě metoda `GenresMenu` kontroléru `NavigationController`, a poté zobrazí výsledek, který jí dětská akční metoda vrátila.

Kdybych nyní aplikaci spustil, na stránce by se mi zobrazila hlavička s nadpisem `Library` a dva panely. Na levé straně seznam žánrů knih a na pravé tabulka se všemi knihami v databázi. Když ale budu klikat na jednotlivé žánry, budou se mi v tabulce stále zobrazovat všechny knihy, bez ohledu na to, jakého žánru jsou. To proto, že jsem ještě neupravil akční metodu `List` kontroléru `BookController` tak, aby brala v potaz zvolený žánr. Metodu upravím, jak je zobrazeno ve výpisu 4-22.

Výpis 4-22: Upravená metoda `List` kontroléru `BookController`. *Zdroj: vlastní zpracování*

```
public ViewResult List(string genre)
{
    IEnumerable<Book> books =
        repository.Books.Where(b => genre == null || b.Genre == genre);
    return View(books);
}
```

Metoda `List` nyní přijímá parametr typu `string`. Hodnota tohoto parametru se sem dostane z pohledu `GenresMenu`, kde ji ukládám v metodě `RouteLink` anonymnímu objektu do proměnné `genre`. Aby se hodnota z proměnné do parametru dostala, musí být stejně pojmenované a být stejného typu (nebo aspoň musí být předávaná hodnota schopna implicitního přetypování na typ parametru). Z repositáře se poté získají pouze ty knihy, které mají požadovaný žánr. Pokud parametr nemá hodnotu (je `null`), z repositáře se získají všechny objekty.

Nyní již filtrování podle žánru funguje, ale bylo by dobré, aby tlačítko zvoleného aktuálně žánru bylo vizuálně odlišené od ostatních. Pohledu `GenresMenu` budu muset předat informaci o tom, na které tlačítko uživatel kliknul. To mohu udělat pomocí objektu ViewBag nebo mohu vytvořit tzv. ViewModel. ViewModel nebude součástí doménové vrstvy, ale vrstvy prezentační. Model bude sloužit jako obalový objekt pro informace využívané

pohledem `GenresMenu`. Ve složce `Library.Presentation/Models` vytvořím třídu `NavigationViewModel`. Ve výpisu 4-23 je zobrazen obsah souboru.

Výpis 4-23: Obsah souboru `NavigationViewModel.cs`. *Zdroj: vlastní zpracování*

```
using System.Collections.Generic;

namespace Library.Presentation.Models
{
    public class NavigationViewModel
    {
        public IEnumerable<string> Genres { get; set; }
        public string SelectedGenre { get; set; }
    }
}
```

Model definuje dvě vlastnosti. První je kolekce, ve které budou uloženy všechny žánry. Druhý bude uchovávat informaci o vybraném žánru. Právě tento model bude akční metoda `GenresMenu` kontroléru `NavigationController` předávat pohledu. Akční metodu tedy upravím, aby objekt `NavigationViewModel` vytvořila a naplnila. Upravená metoda je ve výpisu 4-24.

Výpis 4-24: Upravená metoda GenresMenu. *Zdroj: vlastní zpracování*

```
public PartialViewResult GenresMenu(string genre = null)
{
    NavigationViewModel navigationModel = new NavigationViewModel
    {
        Genres = repository.Books
                    .Select(b => b.Genre)
                    .Distinct()
                    .OrderBy(g => g),
        SelectedGenre = genre
    };
    return PartialView(navigationModel);
}
```

Metoda také nyní přijímá parametr `genre` typu `string`, jehož hodnotu uloží do vlastnosti `Genre` objektu `NavigationViewModel`. Díky konfiguraci routing systému do něj bude automaticky předána hodnota při kliknutí na jedno z tlačítek v panelu.

Nakonec upravím soubor `GenresMenu.cshtml`, aby měl definovaný typ modelu takový, který mu předávám v akční metodě. Také upravím třetí argument volání metody `Html.RouteLink`. Pokud se hodnota link bude rovnat hodnotě vlastnosti `Genre` modelu, připojí se k atributu `class` hodnota `btn-primary`, což způsobí vizuální odlišení tlačítka. Ve výpisu 4-25 je upravený obsah souboru `GenresMenu.cshtml`.

Výpis 4-25: Upravený soubor `GenresMenu.cshtml`. *Zdroj: vlastní zpracování*

```
@using Library.Presentation.Models
@model NavigationViewModel

@Html.ActionLink("All", "List", "Book", null,
    new { @class = "btn btn-block btn-default"})

@foreach (string link in Model.Genres)
{
    @Html.RouteLink(link,
    new {
        controller = "Book",
        action = "List",
        genre = link
    },
    new { @class = "btn btn-block btn-default"
            + (link == Model.SelectedGenre ? " btn-primary" : "")
    })
}
```

4.4.5 Další úprava směrování URL

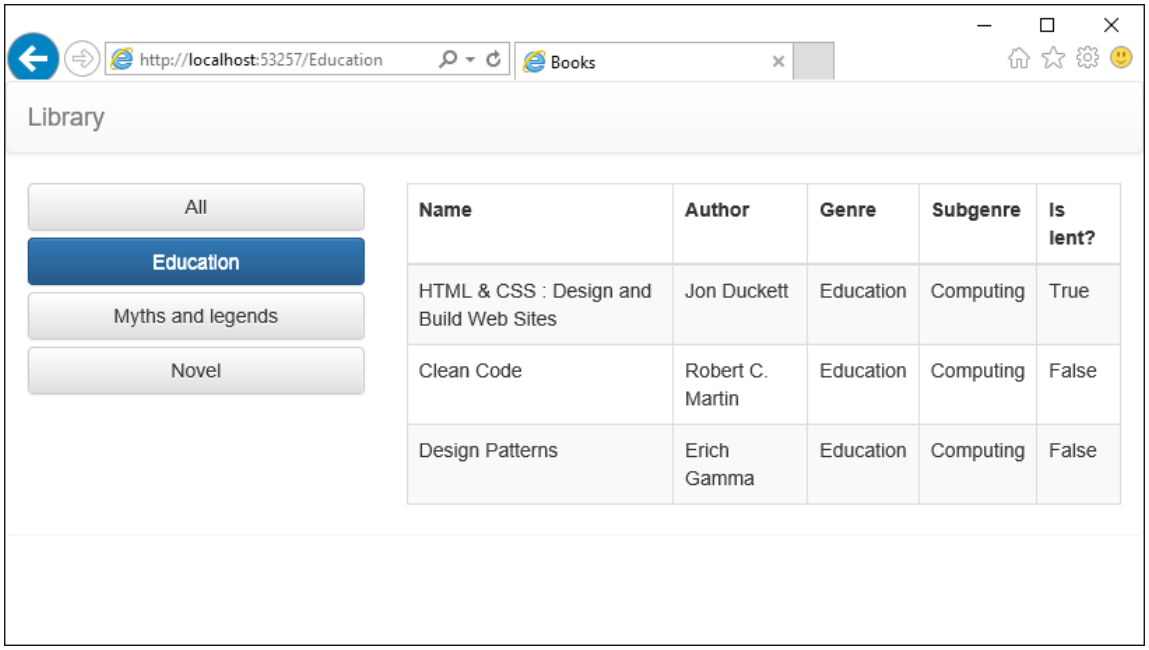
Chtěl bych, aby se URL adresa v prohlížeči zobrazovala v trochu hezčím formátu, než je tomu nyní. Proto znovu upravím metodu `RegisterRoutes` ve třídě `RouteConfig`. Nová podoba metody je ve výpisu 4-26. Touto úpravou docílím toho, že se adresy URL v prohlížeči zobrazují v přehlednějším formátu. Je důležité, aby jednotlivá volání metody `MapRoute` byly přesně v pořadí, v jakém jsou uvedeny ve výpisu 4-26. V opačném případě by se aplikace mohla chovat jinak, než očekávám.

Výpis 4-26: Upravená metoda RegisterRoutes. *Zdroj: vlastní zpracování*

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        null, "",
        new { controller = "Book", action = "List", genre = (string)null}
    );

    routes.MapRoute(
        null, "{genre}",
        new { controller = "Book", action = "List" }
    );

    routes.MapRoute(null, "{controller}/{action}");
}
```

Vytvořil jsem základní prezentační vrstvu. Na hlavní stránce jsem vytvořil hlavičku s nadpisem, která ale v budoucnu může sloužit jako navigační panel, tj. mohu pomocí tohoto panelu přistupovat do jiných částí aplikace. Také se mi na stránce zobrazují v tabulce položky z databázové tabulky `Books`, které je navíc možné filtrovat podle žánru pomocí panelu žánrů. Žánry v tomto panelu se navíc vytvářejí automaticky podle toho, jaké všechny žánry existují u knih v databázi. Na obrázku 4-8 je zobrazena současná podoba stránky v prohlížeči.

Obrázek 4-8: Spuštěná aplikace. *Zdroj: vlastní zpracování*



4.5 Přidávání, úprava a mazání

Mohu zobrazovat seznam knih a filtrovat ho podle žánru. To ale samozřejmě nestačí. Potřebuji, abych mohl pomocí aplikace zobrazovat podrobné informace jednotlivých knih, tyto informace upravovat, a také abych mohl knihy mazat nebo přidávat nové. Tyto čtyři operace se souhrnně označují jako CRUD – Create, Read, Update a Delete. V jisté formě již mám implementovanou operaci Read, kdy načítám jednotlivé položky a některé informace o nich z databáze a zobrazuji je v tabulce. Nyní budu chtít aplikaci upravit tak, abych měl možnost na jednotlivé položky kliknout, načež by okno prohlížeče bylo přesměrováno na stránku s detailními informacemi vybrané knihy. Na stránce knihy budu moci informace upravovat a následně uložit. Také chci mít možnost jednotlivé knihy mazat a přidávat nové.

4.5.1 Úprava vlastnosti knih

Abych mohl vlastnosti knih upravovat, budu potřebovat stránku, ve které se úpravy budou provádět, a také akční metodu, která mi tuto stránku zobrazí. Začnu akční metodou, kterou vytvořím v kontroléru `BookController` a pojmenuji ji `Edit`. Její podoba je ve výpisu 4-27.

Výpis 4-27: Akční metoda `BookController.Edit`. *Zdroj: vlastní zpracování*

```
public ViewResult Edit(int bookID)
{
    Book book = repository.Books
                .FirstOrDefault(b => b.ID == bookID);
    return View(book);
}
```

Metoda přijímá jeden parametr typu `int`, což je identifikátor knihy, jejíž vlastnosti budu chtít upravit. Následně metoda získá podle parametru konkrétní objekt `Book` a předá jej metodě `View`.

Když mám hotovou akční metodu, mohu vytvořit nový pohled, ve kterém budu změny vlastností knihy provádět. Kliknu pravým tlačítkem na složku `Library.Presentation/Views/Book` a v kontextovém menu zvolím Add ➔ `MVC 5 View Page (Razor)`. V okně, které se mi zobrazí, pojmenuji nový pohled `Edit` a kliknu na tlačítko `OK`. Tento pohled bude kompletní HTML stránka. To znamená, že pro něj nepoužiji již existující Layout stránku `_Layout.cshtml`, ale bude se zobrazovat zcela samostatně. Obsah souboru upravím do podoby zobrazené ve výpisu 4-28. V pohledu pomocí `using` bloku a pomocné metody `Html.BeginForm` vytvořím formulář (element `<form>`). Dále ve `foreach` cyklu procházím všemi vlastnostmi modelu (objekt `ViewData.ModelMetaData.Properties`) a vytvářím pro ně editační prvky ve formuláři. Editační prvek samozřejmě nevytvořím pro vlastnost `ID` (identifikátor knihy se nesmí měnit), ale přesto jeho hodnotu ve formuláři musím nějak uchovat. To proto, že při odeslání formuláře zpět serveru, musí akční metoda zpracovávající požadavek znát identifikátor položky, kterou má v databázi aktualizovat. Pro vlastnost `IsLent` vytvořím checkbox samostatně, mimo blok cyklu `foreach`, a to pomocí metody `EditorFor`, která vytvoří element na základě typu objektu.

Výpis 4-28: Obsah souboru Edit.cshtml. *Zdroj: vlastní zpracování*

```
@model Library.Domain.Entities.Book
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.min.css" rel="stylesheet" />
    <title>Book editing</title>
</head>
<body>
    <div class="panel">
        <div class="panel-heading"><h3>Editing book - @Model.Name</h3></div>
        @using (Html.BeginForm()) {
            <div class="panel-body">
                @Html.HiddenFor(m => m.ID)
                @foreach (var property in ViewData.ModelMetadata.Properties) {
                    if (property.PropertyName != "ID"
                        && property.PropertyName != "IsLent") {
                        <div class="form-group">
                            <label>
                                @property.PropertyName
                            </label>
                            @if (property.PropertyName == "Description") {
                                @Html.TextArea(property.PropertyName, null,
                                    new { @class = "form-control", rows = 6 })
                            }
                            else {
                                @Html.TextBox(property.PropertyName, null,
                                    new { @class = "form-control" })
                            }
                        </div>
                    }
                }
                <div class="checkbox">
                    <label>
                        @Html.EditorFor(m => m.IsLent)
                        Book is lent
                    </label>
                </div>
            </div>
            <div class="panel-footer">
                <input type="submit" value="Save" class="btn btn-success" />
                @Html.ActionLink("Cancel", "List", "Book",
                    new { @class = "btn btn-default" })
            </div>
        }
    </div>
</body>
</html>
```

Na konci formuláře jsem ještě vytvořil dvě tlačítka: `Save` a `Cancel`. Tlačítko `Cancel` je ve skutečnosti odkaz směřující na akční metodu `List` kontroléru `Book`. To znamená, že po kliknutí na něj bude prohlížeč přesměrován na úvodní stránku aplikace. Tlačítko `Save` je `<input>` element typu `submit`. Kliknutím na něj se v požadavku předá model akční metodě typu POST, která ho následně předá repositáři, aby ho uložil do databáze.

Nyní provedu malou změnu v pohledu `List` kontroléru `Book`. Chci totiž, aby se kliknutím na název knihy v tabulce zobrazilo editační okno vybrané knihy. Ve výpisu 4-29 je upravený řádek, ve kterém se vytváří název knihy v tabulce. Text s názvem každé knihy bude směřovat na akční metodu `Edit`, které předá svoje ID.

Výpis 4-29: Název knihy jako odkaz. *Zdroj: vlastní zpracování*

```
<td>@Html.ActionLink(@book.Name, "Edit", new { bookID = @book.ID })</td>
```

Když teď spustím aplikaci, otevřu detail nějaké knihy a kliknu na tlačítko `Cancel`, prohlížeč bude přesměrován zpět na úvodní stránku, přesně jak očekávám. Když ale kliknu na tlačítko `Save`, nic se nestane. To proto, že ještě neexistuje žádná akční metoda typu POST s názvem `Edit` (metoda se musí jmenovat stejně jako akční metoda, která vytvořila pohled s formulářem), na kterou by mohl být nasměrován HttpPost požadavek předávající model knihy k uložení. Ale i kdyby taková metoda už existovala, nemohla by upravený model uložit do databáze, protože objekt `EFBookRepository`, který komunikaci s databází zprostředkovává, ještě nemá implementovanou žádnou metodu pro ukládání položek do databáze. Proto obě tyto metody nyní vytvořím.

Začnu implementací metody třídy `EFBookRepository`. Nejdříve metodu přidám do rozhraní `IBookRepository`, jak je zobrazeno ve výpisu 4-30. Metoda bude v parametru přijímat objekt `Book`, který má uložit.

Výpis 4-30: Přidání metody `SaveBook` do rozhraní `IBookRepository`. *Zdroj: vlastní zpracování*

```
void SaveBook(Book book);
```

Po přidání metody do rozhraní ji implementuji ve třídě `EFBookRepository`. Celá metoda `SaveBook` je zobrazena ve výpisu 4-31.

Výpis 4-31: Implementace metody `EFBookRepository.SaveBook`. *Zdroj: vlastní zpracování*

```
public void SaveBook(Book book)
{
    if (book.ID == 0)
    {
        context.Books.Add(book);
    }
    else
    {
        Book dbItem = context.Books.Find(book.ID);
        if (dbItem != null)
        {
            dbItem.Name = book.Name;
            dbItem.Author = book.Author;
            dbItem.Genre = book.Genre;
            dbItem.Subgenre = book.Subgenre;
            dbItem.Description = book.Description;
            dbItem.IsLent = book.IsLent;
        }
    }
    context.SaveChanges();
}
```

Pokud je hodnota vlastnosti `ID` objektu `book` rovna nule, v databázi se vytvoří nová položka, jinak se v databázi aktualizuje položka právě s tímto ID. Když repositář již umí ukládat knihy do databáze, mohu v kontroléru `BookController` vytvořit přetížení metody `Edit`. Nová metoda je zobrazena ve výpisu 4-32.

Výpis 4-32: Přetížení metody `BookController.Edit` s atributem `[HttpPost]`. *Zdroj: vlastní zpracování*

```
[HttpPost]
public RedirectToRouteResult Edit(Book book)
{
    repository.SaveBook(book);
    TempData["bookSaved"]
        = String.Format("The book {0} has been succesfully saved.",
                        book.Name);
    return RedirectToAction("List", "Book");
}
```

Metodu jsem označil C# atributem `[HttpPost]`. Tím jsem docílil toho, že metoda bude zpracovávat pouze požadavky typy POST, a také si nebude server stěžovat, že pro požadavek existují dvě různé metody `Edit`. Uvnitř těla metody zavolám `repository.SaveBook` a předám jí objekt `book` obdržený v parametru. Následně do kolekce `TempData` uložím položku `booSaved`, do které uložím zprávu o úspěšném uložení knihy do databáze. `TempData` je schopna uchovat uložená data i mezi jednotlivými požadavky, takže mohu uloženou zprávu zobrazit později. Nakonec volám metodu `RedirectToAction`. Tato

metoda přesměruje prohlížeč na akční metodu `List` kontroléru `BookController`. To znamená, že po uložení knihy se prohlížeč přesměruje na úvodní stránku. A právě na úvodní stránce budu chtít zobrazit hlášku o úspěšném uložení knihy. Do souboru `Library.Presentation/Views/Book/List.cshtml` vložím kód zobrazený ve výpisu 4-33. Kód vložím dovnitř elementu `<div>` s atributem `class="col-xs-8"` před volání metody `RenderBody`.

Výpis 4-33: Kód pro zobrazení hlášky o úspěšném uložení. *Zdroj: vlastní zpracování*

```
@if (TempData["bookSaved"] != null)
{
    <div class="alert alert-success">@TempData["bookSaved"]</div>
}
```

Nyní již mohu vlastnosti nějaké knihy upravit a pomocí tlačítka `Save` provedené změny uložit. Po úspěšném uložení ze mi v prohlížeči zobrazí úvodní stránka s hláškou o úspěšném uložení. Je zde ale jeden problém. Když jsem vytvářel tabulku `Books` v databázi, všechny sloupečky kromě `Description` jsem definoval jako povinné. To znamená, že ve všech těchto sloupcích jakékoliv položky musí být uložena nějaká hodnota. Kdybych tedy při úpravě vlastností knihy nějaké pole (kromě pole `Description`) nechal prázdné a kliknul na tlačítko `Save`, aplikace by se snažila uložit do sloupce v databázi `null` a ohlásila by chybu. Tento problém vyřeším validací modelu. Ve třídě `Book` vytvořím u všech vlastností, jejichž protějšky v databázi jsou povinné, atribut `[Required]`. Tímto se kontroluje, zda vlastnosti mají nějakou hodnotu, již na úrovni modelu. Není tedy možné předat z formuláře akční metodě model s nenaplněnou vlastností. U každého atributu také definuji chybovou hlášku, která se má zobrazit, pokud pole zůstane prázdné. Upravená metoda je zobrazena ve výpisu 4-34. Budu také muset upravit akční metodu `Edit` kontroléru `Book` (tu která přijímá POST požadavky), aby se nesnažila o uložení do databáze, pokud se v modelu vyskytla chyba. Nová podoba metody je výpisu 4-35.

Výpis 4-34: Upravená třída `Book.cs`. *Zdroj: vlastní zpracování*

```
using System.ComponentModel.DataAnnotations;

namespace Library.Domain.Entities
{
    public class Book
    {
        public int ID { get; set; }

        [Required(ErrorMessage = "You must enter name of the book.")]
        public string Name { get; set; }

        [Required(ErrorMessage = "You must enter authors name.")]
        public string Author { get; set; }

        [Required(ErrorMessage = "You must enter genre.")]
        public string Genre { get; set; }

        [Required(ErrorMessage = "You must enter subgenre.")]
        public string Subgenre { get; set; }

        public string Description { get; set; }
        public bool IsLent { get; set; }
    }
}
```

Výpis 4-35: Upravená metoda `Edit` kontroléru `Book`. *Zdroj: vlastní zpracování*

```
[HttpPost]
public ActionResult Edit(Book book)
{
    if (ModelState.IsValid)
    {
        repository.SaveBook(book);
        TempData["success"]
            = String.Format("The book {0} has been succesfully saved.",
                            book.Name);
        return RedirectToAction("List", "Book");
    }
    else
    {
        return View(book);
    }
}
```

V metodě nyní kontroluji, zda je předaný model validní, tj. zda vlastnost `ModelState` neobsahuje žádné chyby. Právě do této vlastnosti se uloží chyba, pokud je při volání akční metody nějaká vlastnost modelu, označená atributem `[Required]`, prázdná. Pokud se žádná chyba nevyskytne, model se uloží a zavolá se metoda `RedirectToAction`, stejně jako předtím. Pokud se ale nějaká chyba vyskytne, v prohlížeči zůstane zobrazena editační stránka. Když se tak stane, budu chtít, aby se u nevyplněných polí zobrazily chybové hlášky,

které jsem definoval v atributech `[Required]` ve třídě `Book`. Proto do kódu pohledu `Edit.cshtml` přidám volání metody `ValidationMessage`. Volání metody vložím do těla cyklu `foreach` tak, aby se hláška na stránce zobrazila pod textovým polem. Ve výpisu 4-36 je celý vložený kód s voláním metody.

Výpis 4-36: Volání metody `ValidationSumary`. *Zdroj: vlastní zpracování*

```
@Html.ValidationMessage(property.PropertyName)
```

Pokud pro vlastnost, jejíž název je metodě předán, existuje chyba, tato metoda ji zobrazí. V opačném případě se nezobrazí nic.

Chybové hlášky se mi nyní zobrazují a aplikace mě dokonce knihu nenechá uložit, pokud nejsou povinná pole vyplněná, ale bylo by dobré hlášky nějak zvýraznit, aby si jich uživatel vůbec všiml. Také by bylo vhodné, kdyby se zvýraznilo samotné nevyplněné pole. Proto vytvořím nový CSS soubor, ve kterém definuji styly pro chybové hlášky a pro pole, u kterých se chyba vyskytla. Soubor vytvořím ve složce `Library.Presentation/Content` a pojmenuji ho `CustomErrorStyles`. Obsah souboru upravím podle výpisu 4-37. Na soubor se styly samozřejmě musím v elementu `<head>` pohledu `Edit` přidat odkaz.

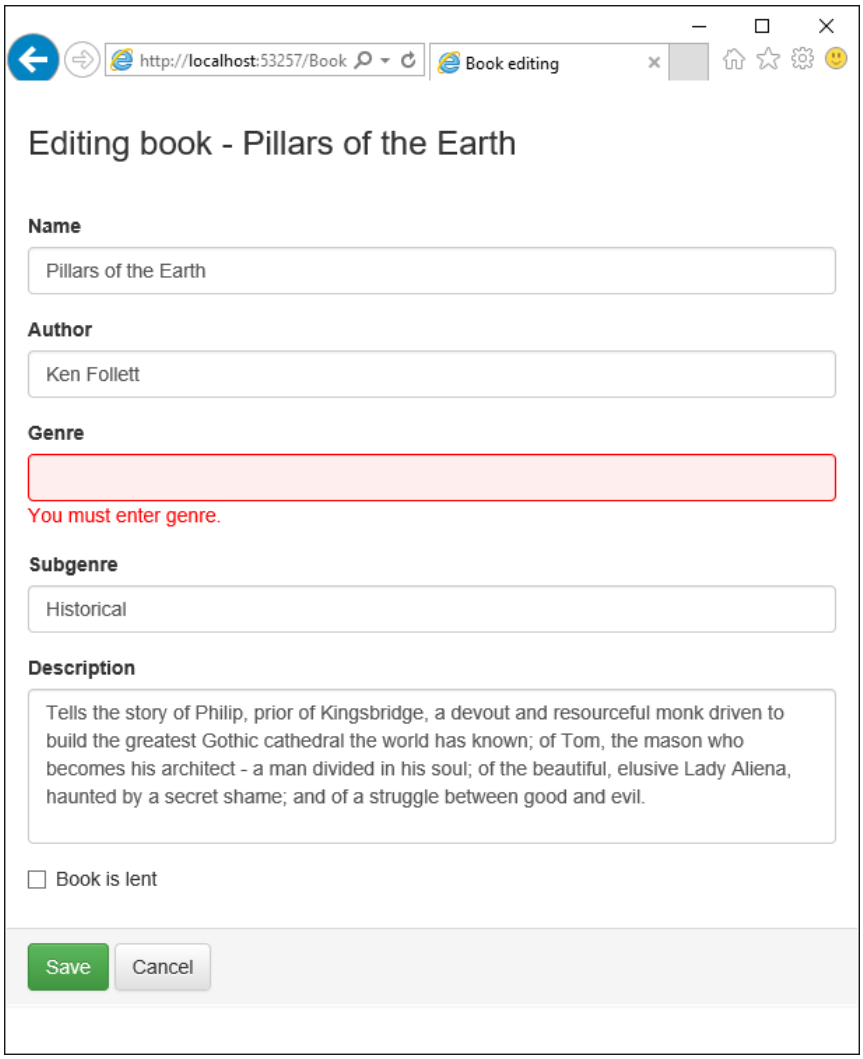
Výpis 4-37: Kaskádové styly pro chybové hlášky a pole. *Zdroj: vlastní zpracování podle Freemana (2013)*

```
.field-validation-error {color: #f00;}
.input-validation-error {border: 1px solid #f00; background-color: #fee;}
```

Nyní je editace existujících položek databáze plně funkční. Kliknutím na jednotlivé položky mohu zobrazovat detaily jednotlivých knih, upravovat je a následně změny uložit zpět do databáze. Na obrázku 4-9 je zobrazena podoba editačního formuláře i s některými chybovými hláškami.

Zde je vhodné zmínit, že validace toho, že jsou pole vyplněna, by se měla provádět ještě dříve, než se formulář odešle na server. To se dá provést pomoci Javascriptu, takže se kontrola polí provede přímo v prohlížeči. Pokud je nějaké pole nevyplněné, Javascript kód zabrání odeslání formuláře na server. Stále je ale dobré implementovat validaci i na straně serveru, může se totiž stát například to, že uživatel bude mít v prohlížeči podporu Javascriptu vypnutou. V tom případě by se serveru odeslal i špatně vyplněný požadavek a bez validace na straně serveru by se aplikace pokusila o uložení špatně naplněného objektu do databáze, což by mělo za následek chybu.

Obrázek 4-9: Editační formulář. *Zdroj: vlastní zpracování*



4.5.2 Přidávání nových knih

Uživateli musím umožnit, aby v aplikaci mohl přidávat do databáze nové knihy. V kontroléru `Book` vytvořím novou akční metodu `Create`. Její tvar je zobrazen ve výpisu 4-38.

Výpis 4-38: Akční metoda `Create`. *Zdroj: vlastní zpracování*

```
public ViewResult Create()
{
    return View("Edit", new Book());
}
```

Uvnitř metody volám metodu `View`, pro kterou specifikuji, který pohled má vytvořit a vrátit. Metoda tedy vrátí ten samý pohled `Edit`, který vrací akční metoda `Edit` (ta, která se

zavolá při kliknutí na název knihy v tabulce), a předá tomuto pohledu nenaplněný objekt `Book`. Pohled tedy bude mít nastavený model knihy, ale všechna pole budou prázdná.

Po této změně ale budu muset zmíněný pohled upravit. Protože jsem ve volání pomocné metody `Html.BeginForm` předtím nespecifikoval, na který kontrolér a akční metodu má být prohlížeč při odeslání formuláře přesměrován, bude přesměrován vždy na akční metodu, která pohled s formulářem vytvořila. Když ale pohled vytvoří metoda `Create`, při odeslání formuláře bude požadavek přesměrován zpět na tuto metodu. Já ale chci, aby byl požadavek nasměrován na POST akční metodu `Edit`. Upravím proto volání metody `Html.BeginForm`, jak je zobrazeno ve výpisu 4-39. Pomocí parametrů specifikuji, na který kontrolér a akční metodu má být požadavek při odeslání formuláře vždy přesměrován.

Výpis 4-39: Řádek s upraveným voláním metody `Html.BeginForm`. *Zdroj: vlastní zpracování*

```
@using (Html.BeginForm("Edit", "Book")) {
```

Nakonec ještě někam do pohledu `List` umístím tlačítko `Add book`, což bude odkaz na akční metodu `Create`. Současně s tím poupravím celý pohled `List`. Upravený kód pohledu je ve výpisu 4-40. Ve výpisu jsem neuvedl obsah elementu `<table>`, jelikož podobu tabulky jsem nijak nezměnil.

Výpis 4-40: Upravený soubor List.cshtml. *Zdroj: vlastní zpracování*

```
<div class="panel panel-primary">
    <div class="panel-heading clearfix">
        <div class="pull-right">
            @Html.ActionLink("Add new book", "Create", null,
                new { @class = "btn btn-default" })
        </div>
    </div>
    <div class="panel-body">
        <table class="table table-striped">
            // Tabulka zůstává beze změny
        </table>
    </div>
</div>
```

Nyní již mohu do databáze přidávat nové položky. Při vyplnění formuláře nové knihy a kliknutí na tlačítko `Save` se akční metodě předá objekt `Book` s vlastností `ID` rovné nule. Hodnota této vlastnosti se poté kontroluje v metodě `SaveBook` objektu `EFBookRepository`. A právě když je vlastnost `ID` rovna nule, vytvoří se v databázi nová položka se sloupečky naplněnými hodnotami vlastností modelu.

4.5.3 Mazání existujících knih

Poslední operací, kterou musím implementovat, je operace Delete, tedy mazání existujících knih z databáze. Začnu přidáním nové metody do rozhraní `IBookRepository` a následnou implementací této metody v třídě `EFBookRepository`. Metoda přidaná do rozhraní je zobrazena ve výpisu 4-41. Metoda bude přijímat ID knihy, která se má smazat a bude akční metodě vracet název smazané knihy, proto je návratová hodnota typu `string`.

Výpis 4-41: Přidání metody `DeleteBook` do `IBookRepository`. *Zdroj: vlastní zpracování*

```
string DeleteBook(int bookID);
```

Implementace metody třídou `EFBookRepository` je vidět ve výpisu 4-42.

Výpis 4-42: Implementace metody `DeleteBook`. *Zdroj: vlastní zpracování*

```
public string DeleteBook(int bookID)
{
    Book dbItem = context.Books.Find(bookID);
    if (dbItem != null)
    {
        context.Books.Remove(dbItem);
        context.SaveChanges();
    }
    return dbItem.Name;
}
```

Upravil jsem datovou vrstvu, aby byla schopna mazat položky z databáze. Teď musím upravit prezentační vrstvu, aby uživatel měl možnost knihy skutečně mazat. Nejdříve vytvořím v kontroléru `Book` novou akční metodu `Delete` a potom tlačítko s odkazem na tuto metodu, které se bude zobrazovat u každé položky v tabulce pohledu `List`. Nová akční metoda je zobrazena ve výpisu 4-43.

Výpis 4-43: Implementace akční metody `Delete`. *Zdroj: vlastní zpracování*

```
[HttpPost]
public RedirectToRouteResult Delete(int bookID)
{
    string bookName = repository.DeleteBook(bookID);
    if (bookName != null)
    {
        TempData["bookDeleted"] =
            String.Format("The book \"{0}\" was deleted", bookName);
    }
    return RedirectToAction("List");
}
```

Metodu jsem označil jako POST. To proto, že metody, které způsobují nějaké změny (tato metoda maže položku), by neměly přijímat požadavky typu GET. Kdybych metodu atributem `[HttPost]` neoznačil, mohlo by docházet k mazání položek, i když si to uživatel pomocí tlačítka explicitně nevyžádal. Uvnitř metody se volá metoda `repository.DeleteBook`, která vrací název smazané knihy. Následně je do `TempData` uložena zpráva o smazání knihy. Tato zpráva by se měla zobrazit v pohledu `List`, kam je prohlížeč metodou `Delete` přesměrován. Kód pro zobrazení zprávy přidám do souboru `_Layout.cshtml` na místo, kde již je kód pro zobrazení zprávy o úspěšném uložení knihy. Ve výpisu 4-44 je kód zobrazen.

Výpis 4-44: Kód pro zobrazení zprávy o smazání knihy. *Zdroj: vlastní zpracování*

```
@if (TempData["bookDeleted"] != null)
{
    <div class="alert alert-warning">@TempData["bookDeleted"]</div>
}
```

Jako poslední zbývá vytvořit tlačítko s odkazem na tuto metodu. Tentokrát ale nemohu použít pomocnou metodu `Html.ActionLink`. Kliknutím na odkaz, který metoda `Html.ActionLink` vytváří, se totiž na server odešle požadavek typu GET, a akční metoda, která by měla tento požadavek zpracovat (tedy metoda `Delete`), je označena jako POST (nemůže tedy přijímat požadavky GET). Já tuto situaci vyřeším použitím formuláře a tlačítkem `submit`, který odesílá na server požadavek POST. Do tabulky v souboru `List.cshtml` vložím nový sloupec s prázdnou hlavičkou a do `foreach` cyklu, ve kterém se tvoří jednotlivé řádky tabulky, vložím nový element `<td>`. Obsah elementu je zobrazen ve výpisu 4-45.

Výpis 4-45: Přidání tlačítka `Delete` do řádku tabulky. *Zdroj: vlastní zpracování*

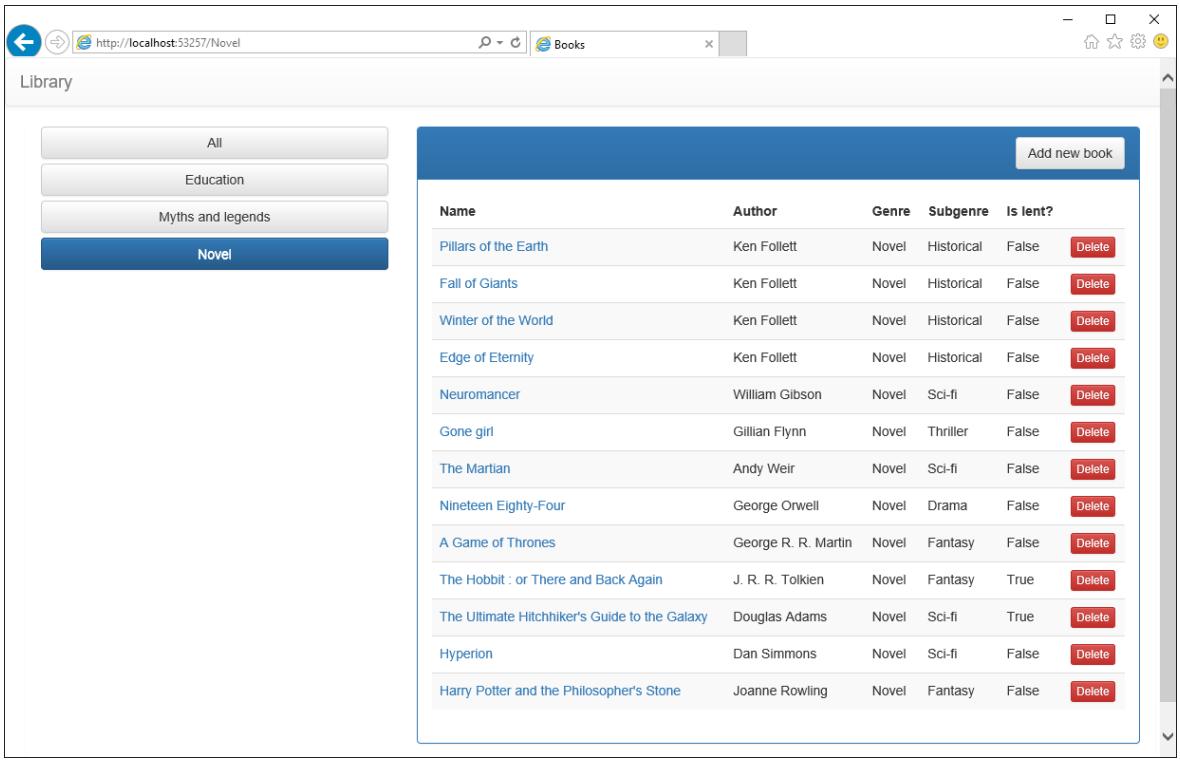
```
<td>
    @using (Html.BeginForm("Delete", "Book"))
    {
        @Html.Hidden("bookID", book.ID)
        <input type="submit"
               class="btn btn-danger btn-xs"
               value="Delete" />
    }
</td>
```

Uvnitř elementu jsem také vytvořil skrytý prvek `bookID`, do kterého se ukládá identifikátor knihy. Tato hodnota je potom předána do parametru akční metody `Delete`.

Nyní mám implementované všechny čtyři CRUD operace. Mohu tedy do databáze přidávat nové knihy nebo existující knihy z databáze mazat. Také si mohu nechat zobrazit detailní informace existujících knih a dokonce tyto informace měnit a ukládat zpět do databáze.

Na obrázku 4-10 je zobrazena finální podoba úvodní stránky aplikace.

Obrázek 4-10: Úvodní stránka aplikace. *Zdroj: vlastní zpracování*



4.6 Psaní unit testů v průběhu vývoje aplikace

Když jsem zakládal řešení `Library`, vytvořil jsem v něm tři projekty. Ve dvou z nich jsem se pohyboval v průběhu celého vývoje aplikace, ale o tom třetím jsem se dosud vůbec nezmínil. Je to projekt `Library.UnitTests`. V tomto projektu jsem i přes to, že jsem se o tom v průběhu vývoje nezmínil, průběžně vytvářel unit testy. Nový unit test je vhodné napsat vždy, když v aplikaci naprogramuji nějaké nové chování, nejčastěji při implementaci nové metody. Avšak kód unit testu může být dlouhý, a také je potřeba jej někdy změnit, pokud se například změnila metoda, jejíž chování unit test testuje. Proto jsem výpisy s unit testy nevkládal přímo do textu práce, ale místo toho jsem je v konečné podobě umístil do přílohy.

5. Výsledky a diskuse

5.1 Zhodnocení

Aplikace je nyní ve stavu, kdy je schopna provádět základní operace nad databází, to znamená čtení, úpravu, vytváření a mazání. Poskytuje uživateli snadno použitelné uživatelské rozhraní a zvolená architektura umožňuje v budoucnu aplikaci snadno upravovat a přidávat nové funkce. K současným komerčním databázovým aplikacím má samozřejmě ještě daleko, ale myslím, že hlavní principy a možnosti a sílu MVC frameworku předvedla dostatečně.

5.2 Možnosti dalšího vývoje

Je toho samozřejmě ještě mnoho, co by se dalo vylepšit a doprogramovat, ale na co již není v této práci prostor. Mezi prvními úpravami by mohla být implementace funkce stránkování. Nyní, s malým počtem knih v databázi, není to, že se všechny zobrazují na jedné jediné stránce, žádný problém. Jak ale budou knihy do databáze přibývat, tabulka s knihami bude na stránce delší a delší a práce s ní bude nepřehledná a nepohodlná. Také by bylo vhodné vylepšit filtrování knih. Pro plnohodnotnou práci je filtrování pouze podle žánru nedostatečné. V aplikaci by mělo být možné filtrovat podle většiny vlastností položek, tj. podle žánru, sub-žánru, jména autora atd. Důležitá je také funkce vyhledávání, kdy uživatel bude moci do textového pole zadat například jméno nebo část jména autora a aplikace mu v tabulce zobrazí všechny vyhovující položky. Další funkcí, která je pro webovou aplikaci důležitá, je identifikace uživatele a kontrola oprávnění, které uživatel má. Vylepšením by se určitě nevyhnula ani samotná databáze. Já jsem všechna data kvůli jednoduchosti a přehlednosti ukládal do jediné tabulky. Pro autory, žánry a sub-žánry by však pravděpodobně v databázi byly vytvořeny vlastní tabulky, aby se dala lépe udržovat konzistentnost dat.

6. Závěr

V této práci byly představeny a popsány vlastnosti a možnosti ASP.NET a ASP.NET MVC frameworků. Byl popsán návrhový vzor *model-view-controller* a postup jeho implementace právě pomocí ASP.NET MVC. Také byly představeny další technologie a frameworky, jako je Razor, jQuery a Ajax , které s vývojem MVC aplikací úzce souvisí a MVC Framework vhodně doplňují. V praktické části práce byl podrobně popsán postup vývoje jednoduché aplikace pracující s databází, ve kterém bylo předvedeno praktické použití ASP.MVC a souvisejících technologií a frameworků. Také byl ukázán způsob vytvoření jednoduché databáze a implementace přístupu k datům pomocí Entity Frameworku a návrhového vzoru *repository*.

Kompletní zdrojový kód aplikace je v podobě *řešení* (solution) Visual Studia dostupný na přiloženém CD. K otevření řešení je potřeba Visual Studio Community, nebo některá placená verze Visual Studioa 2013 nebo Visual Studio 2015, s nainstalovanými komponentami Microsoft SQL Server Data Tools a Microsoft Web Developer Tools.

7. Seznam použitých zdrojů

7.1 Tištěné zdroje

ALBAHARI, Joseph, ALBAHARI, Ben, 2012. *C# in a nutshell*. 5th Edition. Sebastopol: O'Reilly. 1064 s. ISBN 978-1-449-32010-2

CHADWICK, Jess, 2011. *Programming Razor*. Sebastopol: O'Reilly. 120 s. ISBN 978-1-449-30676-2

ESPOSITO, Dino, 2014. *Programming Microsoft ASP.NET MVC*. Third Edition. Sebastopol: O'Reilly. 528 s. ISBN 978-0-7356-8094-4

FOWLER, Martin, RICE, David, FOEMMEL, Matthew, HIEATT, Edward, MEE, Robert, STAFFORD, Randy, 2002. *Patterns of Enterprise Application Architecture*. Boston: Addison Wesley. 560 s. ISBN 0-321-12742-0

FREEMAN, Adam, 2013. *Pro ASP.NET MVC 5*. New York: Apress. 832 s. ISBN 978-1-4302-6529-0

GALLOWAY, Jon. Getting Started. In: GALLOWAY, Jon, WILSON, Brad, ALLEN, K. Scott, MATSON, David, 2014. *Professional ASP.NET MVC 5*. Indianapolis: Wrox press. s. 1-30. ISBN 978-1-118-79475-3

GALLOWAY, Jon. Controllers. In: GALLOWAY, Jon, WILSON, Brad, ALLEN, K. Scott, MATSON, David, 2014. *Professional ASP.NET MVC 5*. Indianapolis: Wrox press. s. 31-47. ISBN 978-1-118-79475-3

GALLOWAY, Jon, HAACK, Phil. Views. In: GALLOWAY, Jon, WILSON, Brad, ALLEN, K. Scott, MATSON, David, 2014. *Professional ASP.NET MVC 5*. Indianapolis: Wrox press. s. 49-74. ISBN 978-1-118-79475-3

WILSON, Brad. Unit Testing. In: GALLOWAY, Jon, WILSON, Brad, ALLEN, K. Scott, MATSON, David, 2014. *Professional ASP.NET MVC 5*. Indianapolis: Wrox press. s. 407-427. ISBN 978-1-118-79475-3

MACDONALD, Matthew, 2012. *Beginning ASP.NET 4.5 in C#*. New York: Apress. 922 s. ISBN 978-1-4302-4251-2

MUELLER, John Paul, 2013. *Microsoft ADO.NET Entity Framework Step by Step*. Redmond: Microsoft Press. 448 s. ISBN 978-0-735-66416-6

PAZ, José Rolando Guay, 2013. *Beginning ASP.NET MVC 4*. New York: Apress. 300 s. ISBN 978-1-4302-5752-3

7.2 Elektronické zdroje

MICROSOFT, nedatováno. *ASP.NET Overview* [on-line]. [cit. 2015-12-31]. Dostupný z WWW: https://msdn.microsoft.com/en-us/library/4w3ex9c2.aspx.

MICROSOFT, nedatováno. Introduction to ASP.NET Web Forms [on-line]. [cit. 2016-02-11]. Dostupný z WWW: http://www.asp.net/web-forms/what-is-web-forms.

8. Přílohy

8.1 Příloha A - Unit testy

Unit test 1. Test získání všech knih z databáze. *Zdroj: vlastní zpracování*

```
[TestMethod]
        public void CanGetAllBooks()
        {
            // Arrange
            // Creating mock repository
            Mock<IBookRepository> mock = new Mock<IBookRepository>();
            mock.Setup(m => m.Books).Returns(new Book[]
            {
                new Book { ID = 1, Name = "N1", Genre = "G1" },
                new Book { ID = 2, Name = "N2", Genre = "G2" },
                new Book { ID = 3, Name = "N3", Genre = "G3" },
                new Book { ID = 4, Name = "N4", Genre = "G2" },
                new Book { ID = 5, Name = "N5", Genre = "G2" },
                new Book { ID = 6, Name = "N6", Genre = "G3" }
            });
            // Creating BookController
            BookController controller = new BookController(mock.Object);

            // Action
            Book[] result = ((IEnumerable<Book>)controller.List(null)
                            .Model).OrderBy(b => b.ID).ToArray();

            // Assert
            Assert.AreEqual(result.Length, 6);
            Assert.IsTrue(result[0].Name == "N1" && result[0].Genre == "G1");
            Assert.IsTrue(result[1].Name == "N2" && result[1].Genre == "G2");
            Assert.IsTrue(result[2].Name == "N3" && result[2].Genre == "G3");
            Assert.IsTrue(result[3].Name == "N4" && result[3].Genre == "G2");
            Assert.IsTrue(result[4].Name == "N5" && result[4].Genre == "G2");
            Assert.IsTrue(result[5].Name == "N6" && result[5].Genre == "G3");
        }
```

Unit test 2. Test získání knih z databáze podle žánru. *Zdroj: vlastní zpracování*

```
[TestMethod]
        public void CanFilterByGenre()
        {
            // Arrange
            // Creating mock repository
            Mock<IBookRepository> mock = new Mock<IBookRepository>();
            mock.Setup(m => m.Books).Returns(new Book[]
            {
                new Book { ID = 1, Name = "N1", Genre = "G1" },
                new Book { ID = 2, Name = "N2", Genre = "G2" },
                new Book { ID = 3, Name = "N3", Genre = "G3" },
                new Book { ID = 4, Name = "N4", Genre = "G2" },
                new Book { ID = 5, Name = "N5", Genre = "G2" },
                new Book { ID = 6, Name = "N6", Genre = "G3" }
            });
            // Creating BookController
            BookController controller = new BookController(mock.Object);

            // Action
            Book[] result = ((IEnumerable<Book>)controller.List("G2").Model)
                            .OrderBy(b => b.ID).ToArray();

            // Assert
            Assert.AreEqual(result.Length, 3);
            Assert.IsTrue(result[0].Name == "N2" && result[0].Genre == "G2");
            Assert.IsTrue(result[1].Name == "N4" && result[0].Genre == "G2");
            Assert.IsTrue(result[2].Name == "N5" && result[0].Genre == "G2");
        }
```

Unit test 3. Test získání jedné knihy podle ID pro editaci. *Zdroj: vlastní zpracování*

```
[TestMethod]
        public void CanEditBook()
        {
            // Arrange
            // Creating mock repository
            Mock<IBookRepository> mock = new Mock<IBookRepository>();
            mock.Setup(m => m.Books).Returns(new Book[]
            {
                new Book { ID = 1 },
                new Book { ID = 2 },
                new Book { ID = 3 },
                new Book { ID = 4 }
            });
            // Creating BookController
            BookController controller = new BookController(mock.Object);

            // Action
            Book book1 = (Book)controller.Edit(2).Model;
            Book book2 = (Book)controller.Edit(3).Model;

            //Assert
            Assert.IsTrue(book1.ID == 2);
            Assert.IsTrue(book2.ID == 3);
        }
```

Unit test 4. Test správného chování při pokusu o získání neexistující knihy pro editaci. *Zdroj: vlastní zpracování*

```
[TestMethod]
        public void CannotEditNonexistentBook()
        {
            // Arrange
            // Creating mock repository
            Mock<IBookRepository> mock = new Mock<IBookRepository>();
            mock.Setup(m => m.Books).Returns(new Book[]
            {
                new Book { ID = 1 },
                new Book { ID = 3 },
                new Book { ID = 4 }
            });
            // Creating BookController
            BookController controller = new BookController(mock.Object);

            // Action
            Book book = (Book)controller.Edit(2).Model;

            // Assert
            Assert.IsNull(book);
        }
```

Unit test 5. Test mazání knihy podle ID. *Zdroj: vlastní zpracování*

```
[TestMethod]
        public void CanDeleteBook()
        {
            // Arrange
            // Create book
            Book book = new Book { ID = 3, Name = "N3", Genre = "G3" };

            // Creating mock repository
            Mock<IBookRepository> mock = new Mock<IBookRepository>();
            mock.Setup(m => m.Books).Returns(new Book[]
            {
                new Book { ID = 1, Name = "N1", Genre = "G1" },
                new Book { ID = 2, Name = "N2", Genre = "G2" },
                book
            });

            // Creating BookController
            BookController controller = new BookController(mock.Object);

            // Action
            controller.Delete(book.ID);

            //Assert
            mock.Verify(m => m.DeleteBook(book.ID));
        }
```

Unit test 6. Test vytvoření žánrů. *Zdroj: vlastní zpracování*

```
[TestMethod]
public void CanCreateGenres()
{
    // Arrange
    // Creating mock repository
    Mock<IBookRepository> mock = new Mock<IBookRepository>();
    mock.Setup(m => m.Books).Returns(new Book[]
    {
        new Book { ID = 1, Name = "N1", Genre = "G3" },
        new Book { ID = 2, Name = "N2", Genre = "G2" },
        new Book { ID = 3, Name = "N3", Genre = "G1" },
        new Book { ID = 4, Name = "N4", Genre = "G2" },
        new Book { ID = 5, Name = "N5", Genre = "G2" },
        new Book { ID = 6, Name = "N6", Genre = "G1" }
    });
    // Creating NavigationController
    NavigationController controller = new NavigationController(mock.Object);

    // Action
    string[] result = ((NavigationViewModel)controller.GenresMenu().Model).Genr
es.ToArray();

    // Assert
    Assert.AreEqual(result.Length, 3);
    Assert.IsTrue(result[0] == "G1");
    Assert.IsTrue(result[1] == "G2");
    Assert.IsTrue(result[2] == "G3");
}
```